

Leveraging Attestation Techniques for Trust Establishment in Distributed Systems

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

in englischer Sprache
zur Erlangung des Grades eines
Doktor der Naturwissenschaften (Dr. rer. nat.)
vorgelegt von

Dipl.-Inform.
Frederic Stumpf
aus Jugenheim



Erstreferent:	Professor Dr. rer. nat. Claudia Eckert
Zweitreferent:	Professor Dr. rer. nat. Uwe Baumgarten
Drittreferent:	Professor Dr. techn. Stefan Katzenbeisser

Tag der Einreichung: 02. November 2009
Tag der Disputation: 21. Dezember 2009

Hochschulkennziffer D17
Darmstadt, 2010

Kurzfassung (Deutsch)

Mit dem Wachstum der Komplexität von Softwaresystemen geht ein Wachstum an aufgedeckten Sicherheitslücken und Fehlern in den Softwaresystemen einher. Einmal ausgenutzt, erlauben es diese Schwachstellen einem Angreifer, bösartige Programme zu installieren, wie etwa Malware oder Spyware, die einzelne Aktionen eines Benutzers, Passwörter, Kreditkarteninformationen, Gebote in Auktionen oder andere sensitive Daten abhören, speichern und weiter verteilen können. Dieses Problem wird weiterhin dadurch verschlimmert, dass eine einzige Schwachstelle in einer einzelnen Anwendung einen Kontrollverlust über das ganze System bewirken kann. Folglich ist es für Benutzer sehr schwer festzustellen, ob das Softwaresystem ihres Computers vertrauenswürdig ist oder nicht. Dieses Wissen ist jedoch notwendig, um Benutzern mehr Sicherheit und damit mehr Vertrauen in den Umgang mit ihrem Softwaresystem zu geben.

In dieser Dissertation werden für diese Problematik Konzepte und Mechanismen entwickelt, um verifizierbare Nachweise zu erstellen, ob sich ein System in einem bestimmten Zustand befindet. Die zugrunde liegende Idee der Arbeit besteht darin, nachzuweisen, ob ein System qua Konstruktion bzw. Konfiguration festgelegte Eigenschaften besitzt, so dass im Vertrauen auf die Gültigkeit dieser Eigenschaften eine Interaktion mit einem System vertrauenswürdig abgewickelt werden kann. Besitzt ein System die spezifizierten Eigenschaften, so kann dies erkannt und eine Interaktion mit dem System rechtzeitig abgebrochen werden. Um diese Anforderungen umzusetzen, werden in dieser Dissertation Konzepte und Mechanismen entwickelt, die auf drei Bausteinen beruhen. Erst die Kombination aller drei Bausteine ermöglicht den sicheren Vertrauensaufbau in ein System.

Der erste Baustein beinhaltet sichere Attestationsprotokolle. Diese Attestationsprotokolle bedienen sich kryptographischer Maßnahmen und ermöglichen es, den Systemzustand eines Softwaresystems vertrauenswürdig zu übermitteln. Als Attestationstechnik werden die Mechanismen eines Trusted Platform Modules (TPM) eingesetzt, um die Authentizität der Eigenschaften eines Systems zu gewährleisten. In dem Kontext dieses Bausteins wird aufgezeigt, welche Anforderungen an Attestationsprotokolle gestellt werden und wie sich sichere Attestationsprotokolle realisieren lassen. Dazu werden mehrere sichere Attestationsprotokolle entwickelt, die unterschiedliche Eigenschaften erfüllen und damit in unterschiedlichen Szenarien eingesetzt werden können. Die entworfenen Protokolle werden bezüglich ihrer Sicherheit sowohl formal als auch informell verifiziert.

Der zweite Baustein ist eine attestationsunterstützende Systemarchitektur, in die sich die Attestationsprotokolle einbetten lassen. Die Systemarchitektur zeichnet sich durch einzelne voneinander isolierte Bereiche aus und ist in der Lage, mittels Attestationsprotokollen ihre Vertrauenswürdigkeit zu belegen. Zur Umsetzung dieses Konzeptes werden Virtualisierungstechnologien benutzt, um inhärente Probleme der zugrunde liegenden Attestationstechnik zu lösen. Dieses Vorgehen ermöglicht die Konstruktion von isolierten, vertrauenswürdigen Ausführungsumgebungen, die gegenüber einer Kompromittierung durch Malware robust sind und daher für sensitive Transaktionen genutzt werden können.

Bei dem dritten Baustein handelt es sich um eine sichere Transaktionssoftware, die es ermöglicht, vertrauenswürdige Transaktionen in verteilten Systemen zu tätigen. Hierzu wird anhand des Beispiels E-Commerce eine Software entwickelt, die sich einer geeigneten Kombination von SmartCard-basierter Authentifikation des Nutzers und einer TPM-basierten Attestation der Softwarekomponenten bedient. Diese Transaktionssoftware nutzt dazu die ersten beiden Bausteine und ist somit robust gegenüber Infektionen durch Malware. Weiterhin implementiert die Transaktionssoftware die vorgestellten Konzepte und Protokolle und ist daher in der Lage, ihre Vertrauenswürdigkeit sowohl gegenüber dem Nutzer als auch einer entfernten Plattform zu belegen.

Abstract (English)

As the complexity of current software systems increases, we see a correlative increase in the number of discovered vulnerabilities. These vulnerabilities, once exploited, allow an attacker to surreptitiously install subversive programs, such as malware and spyware, that can eavesdrop, record and distribute a user's actions, passwords, credit card information, bids in auctions or other sensitive data. Exacerbating this problem is the fact that a single vulnerability in a single application can result in the loss of control of the entire system. As a result, it is difficult for users to ascertain if their computer's software system can be trusted or not. However, such assurances are necessary if users are to become more comfortable in using their software systems.

To alleviate this challenge, we develop concepts and methods to create verifiable proofs with which decisions can be made as to whether a particular system is trusted. For this purpose, the solution proposed in this thesis is based on three main building blocks. Only the combination of all these three building blocks enable overcoming the presented challenges.

The first building block comprises secure attestation protocols. Attestation protocols use cryptographic mechanisms and enable to securely deliver integrity information of the system configuration of a particular (remote) platform. To ensure that the delivered integrity information are authentic, the mechanisms provided by a Trusted Platform Module (TPM) are used. In the context of these building blocks, we show which challenges need to be solved in designing secure attestation protocols. We propose a number of different attestation protocols that are adapted to different scenarios and enable establishing trust in a remote entity's platform configuration. We also evaluate our proposed protocols in terms of security and performance. To this end, we formally analyzed our proposed protocols using a model checker and implemented all protocols to gain performance data.

The second building block is a security architecture for non resource-constrained computer systems. This security architecture is based on virtualization techniques and is adapted to efficiently use attestation techniques. It provides an isolated security environment where confidential data can be processed. We also give details about our performed implementation.

The third building block is a secure transaction software that facilitates making secure transactions in distributed systems. The transaction software uses the mechanisms of the first two building blocks and is, thus, resistant against infection from malware. In addition, it implements the proposed concepts and protocols of the first two building

blocks. Hence, it is able to prove its own trust level to a remote platform and to the user of the secure transaction software.

Acknowledgements

The past three years that I spent as a member of the IT security research group at Technische Universität Darmstadt have been very exciting, challenging, interesting and very rewarding. During that time I crossed paths with many wonderful people from all over the world.

First and foremost, I would like to express my deepest gratitude to Prof. Dr. Claudia Eckert for her supervision, unfailing enthusiasm, never-ending support, and respectful professionalism in guiding me through the countless difficult times that I have encountered during my research. In particular, I thank her for the excellent opportunity of being a member of her IT security research group. Moreover, I would like to thank her for the relaxed and co-operative working atmosphere we had in our research group.

I am also very grateful to Prof. Dr. Uwe Baumgarten for his advice and his support as well as for being the co-referent of this thesis.

Very special thanks also to Prof. Dr. Stefan Katzenbeisser who helped me a lot by proof-reading this thesis and for all the fruitful discussions we had together. These discussions improved the quality of this thesis and were especially during the last stage of writing very encouraging.

I am deeply grateful to all other members and former members of the IT security group for their invaluable support, friendship, and encouragement. These were – in alphabetical order – Michael Benz, Thomas Buntrock, Thorsten Clausius, Lars Fischer, Christoph Krauß, Sascha Müller, Taufiq Roachaeli, Patrick Röder, Christian Schneider, Thomas Stibor, Omid Tafreschi, and Henny Walter. I think we all had a great time not only during the fruitful discussions, but also in our spare freetime.

Last but not least, I want to thank my parents, my sister Julia, and the rest of my family for their great support during all the years of my academic and non-academic life. This thesis would not have been possible without them. I dedicate this thesis to them.

Declaration

These doctoral studies were conducted under the supervision of Prof. Dr. Claudia Eckert. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Computer Science at Technische Universität Darmstadt as a candidate for the degree of Doktor der Naturwissenschaften (Dr. rer. nat.). This work has not been submitted for any other degree or award in any other university or educational establishment.

München, March 8, 2010

Contents

Table of Contents	viii
List of tables	1
I Introduction	1
1 Overview	3
1.1 Motivation	3
1.2 Problem Statement	5
1.3 Contribution	6
1.4 Outline of this Thesis	8
2 Background	11
2.1 Trusted Computing	11
2.2 CPU Privilege Levels	13
2.3 Virtualization	14
2.4 Formal Verification of Protocols	15
2.4.1 Theorem Proving	15
2.4.2 Model Checking	15
2.4.3 Comparison of selected tools for protocol analysis	16
2.5 Notation and Definitions	16
2.5.1 Cryptographic Notation	16
2.5.2 Definitions	17
II Attestation Protocols	19
3 Secure Attestation	21
3.1 Introduction	21
3.2 Classification of Attestation Techniques	22
3.2.1 TPM-Based Binary Attestation	22
3.2.2 Software-Based Attestation	23
3.2.3 Property-Based Attestation and Semantic Attestation	24

3.3	Masquerading Attacks on Attestation protocols	24
3.3.1	Integrity Reporting Protocols	25
3.3.3	Masquerading Attacks on Attestation Protocols	26
3.3.4	Attestation over Secure Channels	27
3.3.6	Analysis of the Attack	30
3.4	Evaluation of Possible Solutions	31
3.4.1	Verifying Attestation Challenges	31
3.4.2	Using Shared Secrets	32
3.4.3	Using <i>AIK</i> as Session Key	32
3.4.4	Using Network Monitors	32
3.4.5	Conclusion	32
3.5	Secure Attestation Channels	32
3.5.1	Preventing Masquerading Attacks	33
3.5.3	Alternative Solution	35
3.6	Security Analysis	35
3.6.1	Informal Security Analysis	35
3.6.2	Formal Security Analysis	37
3.7	Attestation as TLS extension	43
3.7.1	Trusted-TLS with Anonymous User	43
3.7.3	Trusted-TLS with User Authentication	45
3.8	Security Analysis	47
3.8.1	Informal Security Analysis	47
3.8.2	Formal Security Analysis	48
3.9	Related Work	50
3.10	Summary	51
4	Scalable Attestation	53
4.1	Introduction	53
4.2	Evaluation of Existing Proposals	54
4.3	Assumptions and Notations	55
4.4	Scalable Solutions	56
4.4.1	Multiple-Hash Attestation	57
4.4.3	Timestamped Hash-Chain Attestation	60
4.4.5	Tickstamp Attestation	63
4.5	Security Analysis	67
4.5.1	Security of the Authentication	67
4.5.2	Formal Security Analysis	68
4.5.3	Security Considerations of the Multiple-Hash Attestation	69
4.5.4	Security Considerations of the Timestamped Hash-Chain Attestation	69
4.5.5	Security Considerations of the Tickstamp Attestation	69
4.6	Evaluation	70
4.6.1	Performance Evaluation	70
4.6.2	Comparison of the protocols	71

4.7	Summary	72
5	Privacy-Preserving Attestation	73
5.1	Introduction	73
5.2	Notation	75
5.3	Ticket-Based Attestation	76
5.3.1	High-Level Description	76
5.3.2	Integrity Reporting Algorithm	77
5.3.3	Integrity Validation Algorithm	79
5.3.4	Performance Optimizations	80
5.3.5	Integrity Verification	82
5.4	Security Mechanisms	83
5.5	Security Analysis	85
5.5.1	Certification Phase	85
5.5.2	Attestation Phase	86
5.6	Summary	87
6	Lightweight Attestation	89
6.1	Introduction	89
6.2	Setting	90
6.2.1	Scenario: Wireless Sensor Networks	90
6.2.2	Scenario: Embedded Systems	91
6.3	Assumptions	92
6.4	Attestation Protocols	94
6.4.1	Periodic Broadcast Attestation Protocol (PBAP)	95
6.4.3	Individual Attestation Protocol (IAP)	97
6.5	Analysis	101
6.5.1	Security Discussion	101
6.5.2	Performance Analysis	103
6.6	Implementation	105
6.7	Summary	106
III	Attestation-supporting Security Architecture	107
7	Security Architecture	109
7.1	Introduction	109
7.2	Security Challenges	110
7.2.1	Malware Attack Points	110
7.2.2	Attacker Model	112
7.3	Ensuring System Integrity using Attestation Techniques	112
7.4	Design Concept of the Security Architecture	113
7.4.1	Overview of the Trusted VMM	114
7.4.2	Overview of the Trust Anchor	115

7.4.3	Virtual Machines	115
7.5	Security Architecture	116
7.5.1	Trusted Virtual Machine Monitor and Management VM	116
7.5.2	Trusted Virtual Machine	118
7.5.3	Virtual TPM	119
7.5.4	Secure Storage	122
7.6	Attestation of Virtual Machines	124
7.7	Evaluation of the Architecture	126
7.7.1	Protection Layers	126
7.7.2	Security Evaluation	128
7.8	Implementation	133
7.9	Related Work	135
7.9.1	Security Kernels	136
7.9.2	Virtual Machine Monitors	136
7.10	Summary	137
8	Virtualization-Enhanced TPMs	139
8.1	Introduction	139
8.2	Virtualizing the TPM	140
8.2.1	Requirements	140
8.2.2	Our Approach	141
8.3	TPM Architecture	142
8.3.1	TPM Protection Rings	143
8.3.2	TPM Back-End Device Driver	143
8.3.3	TPM Control Structure	145
8.3.4	Extended Instruction Set	146
8.4	Direct Native Execution	146
8.4.1	Handling Sensitive Instructions	147
8.4.2	Scheduling the TPM	148
8.5	Secure TPM Context Migration	149
8.6	TPM Credentials	150
8.6.1	Spawning a TPM Context	151
8.6.2	Exiting a TPM Context	152
8.7	Satisfied Security Requirements	153
8.8	Related Work	153
8.9	Summary	154
IV	Transaction Software	155
9	Secure Transaction Software	157
9.1	Introduction	157
9.2	Overall E-Commerce Architecture	159
9.2.1	Client Architecture	159

9.2.2	Server Architecture	160
9.2.3	Ticket-Based Attestation	160
9.3	Trusted Transaction Software	161
9.3.1	Attestation Protocol	161
9.4	User Authentication and Ticket Presentation	164
9.4.1	Password-Based User Authentication and Ticket-Presentation . .	164
9.4.3	Authentication Based on Smart Cards	166
9.5	Security Analysis	166
9.5.1	Forging a Trusted System Configuration	166
9.5.2	Accessing Confidential User Data	166
9.5.3	Security of the Password-Based User Authentication	167
9.6	Implementation	168
9.7	Related Work	170
9.8	Summary	171
10	Signature Creation with TPMs	173
10.1	Introduction	173
10.2	Testing SigG Conformity of the TPM	174
10.2.1	SigG Conformity of the TPM	174
10.2.2	Results	176
10.3	Qualified Electronic Signatures	176
10.3.1	Identification	176
10.3.2	Issuing Qualified Certificates	176
10.4	Trusted Software	178
10.4.1	Trusted Initializing Software	178
10.4.2	Trusted Signing Software	179
10.4.3	Knowledge and possession	179
10.5	Summary	180
V	Summary	183
11	Conclusions	185
VI	Appendix	189
A	AVISPA Sources	191
A.1	Protocol 3.6.3	191
A.2	Protocol 3.7.4	194
	References	218

List of Figures

2.1	CPU privilege levels of the x86 architecture	13
2.2	Summary of selected tools for protocol analysis	16
2.3	A transferred signed message including transmittal of the message in plaintext	17
2.4	A transferred signed message	17
3.1	Masquerading attack on the integrity reporting protocol	27
3.2	Masquerading attack on the TLS-secured remote attestation	28
3.3	State machine of Client (platform B) and Server (platform A).	49
4.1	Delay of simultaneously arriving attestation requests	56
4.2	Tri-state ring buffer of the Multiple-Hash Attestation	59
4.3	Protocol flow of the Timestamped Hash-Chain Attestation	61
4.4	Measured latencies of selected Integrity Reporting Protocols	71
5.1	Distributed Integrity Validation Architecture	74
5.2	Hierarchy of the SML	77
6.1	Cluster of a Wireless Sensor Network	91
6.2	Communication between the TPM and an attestation service to realize the initialization phase of the IAP	105
7.1	Basic design concept of the security architecture	114
7.2	The proposed security architecture	117
7.3	Actions taken when a Trusted VM spawns	119
7.4	Mapping of the PCR values	120
7.5	Binding the vTPM to TPM	121
7.6	Protocol steps of a spawning vTPM that is protected by the hardware TPM	124
7.7	Simplified attestation process	125
7.8	Protection architecture organized in layers	126
7.9	Attack vectors on our security architecture	129
7.10	Implemented Architecture using Xen. The arrows indicate the commu- nication path between the different components.	134

8.1	Layout of a multi-context TPM	142
8.2	Privilege level of a multi-context TPM	143
8.3	Transition between different TPM contexts	144
8.4	TPM Control Structure	145
8.5	Workflow of retiring one VM/TPM and activating the next VM/TPM .	149
8.6	High-Level Migration Protocol	150
8.7	Exiting and Resuming a TPM context (CR of the TPM is set to 0) . . .	151
8.8	Creating a new TPM context (CR of the TPM is set to 0)	152
9.1	Transaction Model	159
9.2	The secure transaction software runs in the TVM of our proposed security architecture	160
9.3	Man-in-the-Middle Attack on Protocol 9.4.2	167
9.4	The spawning of a Trusted VM. The upper shell shows the bootstrap procedure of the TVM. The lower shell shows the contents of the vTPM. In this realization, the TVM uses the x-server of the Management VM. .	168
9.5	Bootstrap procedure of the STS. After starting the TVM, the STS is di- rectly executed. The picture shows the user's stored secret as deposited inside the TVM. The shell also shows the operations of the vTPM- Manager running in the Management VM.	169
9.6	Started STS. After successful validation of the platform configuration, the TVM starts the STS.	170
10.1	Protocol for certificate issuance	177

Part I

Introduction

Chapter 1

Overview

This chapter serves as an overview. We will give a motivation for the topic and formulate the problem statements of the thesis. In this context, we will show why we need attestation techniques for trust establishment and why typical operating systems are not appropriate for supporting these techniques. Moreover, we will shortly summarize the main contribution of this thesis. This chapter additionally gives a short outline of how the thesis is organized.

1.1 Motivation

Current software systems are increasingly used for online transactions. Scenarios where this is done include e-commerce, online banking, or e-business. In such transactions, the software systems are often entrusted with sensitive data, such as personal or financial information. Sensitive data has a very high protection level and unauthorized access to this data must be prevented. One mechanism to protect sensitive data while they are sent over a communication channel is to protect them through the use of cryptographic protocols. However, cryptographic protocols can only secure communication channels and are basically overstrained if an attack is performed at the end of the communication channel, where confidential data is often available in plaintext.

One component that is responsible for protecting sensitive data at the endpoint of a communication channel is the operating system, which provides basic protection mechanisms. These protection mechanisms do not only include establishing cryptographic channels, but also ensure that only authorized processes are allowed to access particular data. For this purpose, the operating system provides for each process an own virtual address space. The virtual address space is transformed using a Memory Management Unit into an address which points to the actual physical location in memory. Unless the operating systems marks a specific area of the address space as shared, a process can only access its own allocated address space and no address space which is allocated by another process.

However, operating systems are very complex and not very reliable. It has been shown that an operating system contains between six and 16 bugs per 1,000 lines of

executable code [15]. A conservative estimate of six bugs per 1,000 lines of code indicates that Windows XP, with a code size of around 50 million lines of code [166], has approximately 300,000 bugs. It should also be noted that typically about 70 percent of the operating system consist of device drivers [167]. These device drivers have error rates that are three to seven times higher than ordinary code [35]. If these errors can be exploited by an attacker, a security vulnerability emerges that may allow an attacker to violate the integrity of the operating system [100]. Since the operating system typically runs in the highest CPU privilege level, an exploited vulnerability can allow an attacker to gain control of the entire system. The attacker is then able to surreptitiously install subversive programs, such as malware and spyware, that can eavesdrop, record and distribute a users' action, passwords, credit card information, bids in auctions or other sensitive data. Exacerbating this problematic exploitation, a single vulnerability in a single application can result in the loss of control of the entire system.

One solution for overcoming this vulnerability is using attestation techniques as defined by the Trusted Computing Group (TCG) [179]. These techniques can be used to detect a trusted and malware free platform if a Trusted Computing-enhanced operating system would be available. To enable a platform to attest to its platform configuration, a measurement agent of a Trusted Computing-enhanced operating system measures every code fragment before execution and stores the resulting values in a protected and shielded location of a Trusted Platform Module (TPM). These values can then be transferred to a remote entity (or a particular user) in order to prove that a specific platform behaves as specified, i.e., it is trusted. However, these mechanisms cannot be realized in currently available operating systems, mainly due to the enormous system complexity and the fact that all components need to be trusted even if this component has no impact on the target application.

Virtualization techniques [69] provide a mechanism for adapting a compartmentalization to operating systems. The main idea of these techniques is to use a hypervisor or virtual machine monitor, which establishes several strong isolated execution environments, that cannot influence one another. A fault or a vulnerability in one environment cannot inflict damage to other environments. It has already been formally shown by Madnick and Donovan [105] that this approach provides substantially better software security and reliability. However, these techniques on their own are no panacea, since they do not provide assurances that a component, e.g. the hypervisor, behaves as specified. A faulty hypervisor could, for example, inspect every instruction issued by a virtual machine and undermine the protection architecture. To make matters worse, an adversary could foist a hypervisor on a user's platform [93] without any notice by the user. This attack vector can also exploit characteristics of virtualization-supporting hardware [60, 135], showing that it is expected that future malware generation use these vectors. This would induce a completely new malware attack model causing future generations of malware to be even more harder to detect than current malware generations. These properties show that virtualization and attestation techniques depend on each other and only both together are a strong foundation for secure systems.

In this thesis, we will show how an appropriate combination of attestation techniques and virtualization technologies can significantly enhance system security. Using

virtualization and attestation, a user can be confident that his platform can be trusted and that his platform has not been compromised by malicious software components. In addition, isolated compartments can be constructed that are immune from infection by other software components and that allow for the construction of robust and self-protecting systems that can hardly be compromised by an adversary.

1.2 Problem Statement

Below, we identify the main challenges that need to be solved in order to use attestation techniques in distributed systems in general, and operating systems in particular.

Secure and Efficient Attestation Channels Measuring system components using the Trusted Platform Module and transferring the obtained measurements to a remote entity is a tedious task and provides several challenges [70, 164, 156, 160]. These challenges include determining how to establish a secure channel between the verifying entity and the prover [164], how to secure this channel and how to efficiently realize it so as to satisfy scalability issues [160]. If this *attestation channel* is not established in a secure manner, an attacker might be able to relay the attestation challenge to another host and then masquerade his host as being in a conforming state [164].

Attestation techniques can be used to ensure the trust level of a remote entity before transferring confidential data to it. However, in many scenarios, e.g., e-commerce, it is important to provide the user of a particular platform with assurances that his own platform has not been tampered with and is in a trusted state. Since a successful attestation could have also been simulated [158], it is important to provide mechanisms for determining how the trust can be guaranteed in a way that the user is convinced that his platform is trusted.

Attestation Complexity Currently available operating systems are not appropriate for supporting attestation techniques [156]. This is due to the enormous system complexity of open and basically non resource-constrained operating system environments and the fact that the process of attestation is basically inefficient [156, 20, 137, 87, 83]. In addition, due to the type of measurement process, not all software components that influence the runtime behavior of an operating system are measured. As a consequence, new operating system environments are required that enable reducing the attestation complexity and are capable of being equipped with means for utilizing attestation techniques.

Privacy Issues The TCG-defined attestation process requires transferring an additional log with which the whole platform configuration is validated. However, transferring this information to a remote platform raises severe privacy concerns [137, 156, 159]. The remote platform is able to discover the full platform configuration, including all running processes, by simply examining the received response. Since this is not, in any case, a desirable feature, privacy-preserving mechanisms are necessary in order to be

able to make a statement about the trust level of a platform without revealing the exact configuration.

1.3 Contribution

This thesis explores the presented challenges, shows how attestation techniques can be applied, and shows that there is no silver bullet solution that solves all challenges. Depending on the scenario, the mechanisms to establish a secure attestation channel vary. This thesis also argues that applying attestation in open and non resource-constrained computer systems requires a system based on virtualization, thus allowing the establishment of several isolated execution environments. We show how such a security architecture can be realized and show that an appropriate combination of attestation and virtualization technologies can significantly enhance system security. Our proposed architecture is adapted to open and distributed non resource-constrained computer systems and provides a means for establishing trust in distributed systems. We will also present an example application for trusted online transactions, which enables purchasing goods on-line and utilizes attestation in combination with virtualization technologies.

In short, the major contributions of this thesis are threefold:

- (1) Derivation of secure attestation protocols that enable establishing secure attestation channels and satisfy different requirements. The attestation protocols have been formally verified to ensure their correctness. These protocols form the first building block for leveraging attestation techniques in distributed systems.
- (2) Concepts and design of a security architecture that enables constructing isolated compartments and supports attestation in order to establish trust in a remote entity's system configuration. The security architecture has also been implemented and the attestation protocols were integrated to show that our approach is feasible. The security architecture is the second building block for leveraging attestation techniques.
- (3) A secure transaction software that utilizes attestation and runs on top of the proposed security architecture. This application has also been implemented and makes the derived trust visible to the user. The secure transaction software forms the last building block of our solution.

In the framework of the research done in this thesis, the following papers have been published:

1. Claudia Eckert, Omid Tafreschi, and Frederic Stumpf. On Controlled Sharing of Virtual Goods. In *7th International Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods*, Nancy, France, September 22, 2009
2. Frederic Stumpf, Benjamin Weyl, Christian Meves, and Marko Wolf. A Security Architecture for Multifunctional ECUs in Vehicles. In *25. VDI/VW-*

Gemeinschaftstagung: Automotive Security, Ingolstadt, Germany, October 19 - 20, 2009

3. Stefan Katzenbeisser, Klaus Kursawe, and Frederic Stumpf. Revocation of TPM Keys. In *Proceedings of the Second International Conference on Trusted Computing (TRUST 2009)*, Oxford, UK, April 6-8, Lecture Notes in Computer Science, pp. 120-132, Springer-Verlag
4. Christian Schneider, Frederic Stumpf, and Claudia Eckert. Enhancing Control of Service Compositions in Service-Oriented Architectures. In *Proceedings of the Third International Workshop on Advances in Information Security (WAIS 2009)*, Fukuoka, Japan, March 16-19, pp. 953-959, IEEE Computer Society,
5. Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the Scalability of Platform Attestation. In *Proceedings of the Third ACM Workshop on Scalable Trusted Computing (ACM STC'08)*, Fairfax, VA, USA, Oct 31, 2008, ACM Press
6. Frederic Stumpf and Claudia Eckert. Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques. In *Proceedings of the Second International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2008)*, Cap Esterel, France, August 25-31, 2008, pp. 1-9, IEEE Computer Society, [Best Paper Award]
7. Frederic Stumpf, Claudia Eckert, and Shane Balfe. Towards Secure E-Commerce Based on Virtualization and Attestation Techniques. In *Proceedings of the Third International Conference on Availability, Reliability and Security (ARES 2008)*, Barcelona, Spain, March 4 - 7, 2008, pp. 376-382, IEEE Computer Society
8. Frederic Stumpf, Lars Fischer, and Claudia Eckert. Trust, Security and Privacy in VANETs - A Multilayered Security Architecture for C2C-Communication. In *23. VDI/VW-Gemeinschaftstagung: Automotive Security*, Wolfsburg, Germany, November 27 - 28, 2007, pp. 55-70, VDI-Verlag
9. Frederic Stumpf, Patrick Röder, and Claudia Eckert. An Architecture providing Virtualization-Based Protection Mechanisms against Insider Attacks. In *Proceedings of the 8th International Workshop on Information Security Applications (WISA 2007)*, Jeju Island, Korea, August 27 - 29, 2007, Lecture Notes in Computer Science, Vol.4867, pp. 142-156, Springer-Verlag
10. Frederic Stumpf, Markus Sacher, Alexander Roßnagel, and Claudia Eckert. The creation of Qualified Signatures with Trusted Platform Modules. *Digital Evidence Journal*, Vol. 4, No. 2, 2007, pp. 81 - 88, Pario Communications Limited
11. Frederic Stumpf, Michael Benz, Martin Hermanowski, and Claudia Eckert. An Approach to a Trustworthy System Architecture Using Virtualization. In *Proceedings of the 4th International Conference on Autonomic and Trusted Computing*

(ATC 2007), Hong Kong, China, July 11-13, 2007, Lecture Notes in Computer Science, Vol.4158, pp. 191-202, Springer-Verlag

12. Christoph Krauß, Frederic Stumpf, and Claudia Eckert. Detecting Node Compromise in Hybrid Wireless Sensor Networks Using Attestation Techniques. *Proceedings of the Fourth European Workshop on Security and Privacy in Ad hoc and Sensor Networks (ESAS 2007)*, Cambridge, UK, July 2007, Lecture Notes in Computer Science, Vol.4572, pp. 203-217, Springer-Verlag
13. Christel Kumbruck, Markus Sacher, und Frederic Stumpf. Vertrauen(skapseln) beim Online-Einkauf. *Datenschutz und Datensicherheit (DuD)*, Mai, 2007, Vieweg-Verlag
14. Frederic Stumpf, Markus Sacher, Alexander Roßnagel, und Claudia Eckert. Erzeugung elektronischer Signaturen mittels Trusted Platform Module. *Datenschutz und Datensicherheit (DuD)*, Mai, 2007, Vieweg-Verlag
15. Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. A robust Integrity Reporting Protocol for Remote Attestation. *Second Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, December, 2006
16. Patrick Röder, Frederic Stumpf, Ralf Grewe, and Claudia Eckert. Hades - Hardware Assisted Document Security. *Second Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, December, 2006

1.4 Outline of this Thesis

This thesis consists of five parts: an introduction (Part 1), three main parts (Part 2 - 4) and a summary (Part 5).

Part 1. Introduction This introduction provides an overview and motivation of the thesis and gives a broad overview of Trusted Computing and virtualization, two techniques that we use heavily. In this section, we formulate the problem statements of the thesis and show why there is a need for applying attestation techniques for trust establishment.

Part 2. Attestation Protocols The second part of the thesis addresses the issue how to realize secure attestation. In this context, we especially look at the challenges that need to be solved in designing secure attestation protocols. We propose a number of different attestation protocols that are adapted to different scenarios and enable establishing trust in a remote entity's platform configuration. We also evaluate our proposed protocols in terms of security and performance. To this end, we formally analyzed our proposed protocols using a model checker and implemented all protocols.

Part 3. Attestation-supporting Security Architecture Part three presents a security architecture for non-constrained computer systems. This security architecture is based on virtualization-techniques and is adapted to efficiently use attestation techniques. It provides an isolated security environment where confidential data can be processed. We also present implementation details about how we implemented our proposal in a proof-of-concept prototype. In this section, we also address the issue of how to virtualize TPMs so that they can be used in security architectures that are based on virtualization. Our proposed security architecture can be used as a foundation for constructing next-generation secure operating systems that rely on Trusted Computing technology.

Part 4. Transaction Software The fourth part presents a secure transaction software that runs on the security architecture introduced in Part 3. The transaction software implements the attestation protocols proposed in Part 2 and also acts as an interface for the user. It enables establishing secure attestation channels with a remote entity and is able to provide the user with assurances that his own platform is trusted. This section also reflects on whether the TPM of a TPM-enhanced platform can be used as a secure signature creation device to provide non-repudiation of concluded agreements.

Part 5. Summary The last part concludes this thesis by summarizing its major contributions.

Chapter 2

Background

This section gives a brief overview of Trusted Computing and virtualization that is important for understanding the approach presented in this thesis. In addition, we will shortly look at the main formal approaches of protocol analysis.

2.1 Trusted Computing

The core component of Trusted Computing is the Trusted Platform Module (TPM) [179]. The TPM can be viewed as functionally equivalent to a high-end smart card that is soldered to the motherboard of a platform. The TPM serves as a *root of trust* within the platform and provides a means of trust establishment using transitive trust. For this purpose, a chain of trust, up to and including the end user applications, is generated with the TPM as trust anchor.

The TPM provides a number of functionalities that include:

- *Protected Capabilities*: The TPM provides a set of exclusive commands with access to protected and shielded locations. These locations provide an area where it is secure to operate on sensitive data. The following TPM functionalities fall into this category:
 - random number generation,
 - registers for recording platform state,
 - secure volatile and non-volatile memory,
 - a SHA-1 hashing engine,
 - asymmetric key generation, encryption and digital signature capabilities.

To reduce the amount of non-volatile memory required inside the TPM, it acts as an access control device for externally stored keys rather than storing all keys itself. Only the storage root key (*SRK*) remains permanently in the TPM and is used to encrypt all other externally stored keys. This strategy offers the same security level as if all keys were stored internally.

This design choice allows to reduce the production costs of a TPM, but introduces the problem that the TPM is not able to reliably destroy externally stored keys once they get compromised. Thus, if an attacker once acquires access rights to a TPM-maintained key and an encrypted version of that key, he can always access that key through the TPM hardware [91].

- *Integrity Measurement and Storing*: Integrity measurement is the process of obtaining metrics of platform characteristics that describe the integrity of a platform. These metrics, or measurements, are held in protected shielded locations, known as Platform Configuration Registers (PCRs). The measurement made by the TPM is written to a specific PCR by combining a hash of a measured software component with the previous value of the PCR. The following function is used to calculate the value for platform configuration register N , where SHA1 refers to the cryptographic hash function used by the TPM and $||$ denotes a concatenation:

$$\text{Extend}(\text{PCR}_N, \text{value}) = \text{SHA1}(\text{PCR}_N || \text{value}). \quad (2.1)$$

In order to establish trust in a computer system, a Trusted Computing enhanced system measures and stores the boot sequence of a platform. When turning on a computer, the *Core Root of Trust Measurement* (CRTM), which resides in the BIOS, is the first component to be executed. The CRTM then measures itself and the BIOS, and then hands over control to the next software component in the boot chain. This process continues for every component involved in the boot sequence of a platform. After execution of the boot sequence, a trusted operating system responsible for measuring each software component before execution. In addition, the trusted operating system records each execution of a software component and generates an entry in the stored measurement log (SML). This SML is maintained by the trusted operating system and enables a later verification of the hash values stored inside the PCRs.

- *Attestation* is the complete process of vouching for the accuracy of information. To attest an entity to its software, i.e., to declare an entity as trusted, the process requires three main mechanisms:
 - *Integrity Measurement* is the process of obtaining the metrics of platform characteristics, i.e., the software's integrity. The obtained metrics are then stored in PCRs. This process can also be understood as a mechanism that exactly identifies which software components are actually running on a specific platform.
 - *Integrity Reporting* is the process of attesting to the content of integrity storage (i.e., PCRs). This process includes delivering the content of integrity storage to a remote entity. To ensure authenticity and freshness of these values, they can be signed with an attestation identity key (*AIK*), which is held in a shielded location of the TPM.

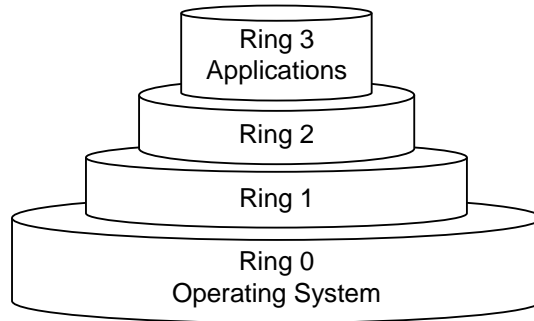


Figure 2.1: CPU privilege levels of the x86 architecture

- *Decision Making* is the process of deciding whether a platform configuration is to be trusted or not. To this end, a verifier processes the received SML and decides based on the PCRs and on reference values, whether a particular platform is trusted or not.

The TPM additionally offers a number of different signing keys. Another important key is the Endorsement Key (*EK*), which is generated by the module manufacturer and injected into the TPM. The *EK* uniquely identifies the TPM and can be used to prove the authenticity of the TPM. In addition, the *EK* is required to obtain an Attestation Identity Key (*AIK*). An *AIK* is generated on a TPM and certified by a trusted third party, typically called a Privacy-CA, or signed via a Direct Anonymous Attestation protocol [23]. With the obtained (proof) certificate, a remote platform can compare the signed values to reference values, in order to check whether or not the platform is in a trusted state.

The TPM also offers a concept called *sealing*, which allows a data block to be bound to a specific platform configuration. A sealed message is created by selecting a range of platform configuration registers, a non-migratable key (i.e., a special type of encryption key that cannot be exported and transferred to another TPM), and the data block that should be sealed. The TPM is then able to decrypt and transfer the sealed data block, only if its current platform configuration matches the platform configuration from the time when the sealing was executed. Sealing provides the assurance that protected messages are only recoverable when the platform is in a known system state.

2.2 CPU Privilege Levels

The x86 architecture provides a basic protection concept with which access to certain resources can be restricted and controlled. This protection concept is realized by a 2-bit privilege level; each level is often referred to as ring. These levels are arranged in a hierarchy from most privileged (ring 0) to least privileged (ring 3). The privilege level determines whether a software component running in one of the rings is allowed

to directly execute privileged instructions, that control basic CPU functionality. Additionally, it controls the accessibility of address-space based on the page tables of the processor. Most x86-software, including operating systems such as Windows and Linux, use only the privilege levels 0 and 3. This is illustrated in Figure 2.1.

2.3 Virtualization

Virtualization introduces a hypervisor called a virtual machine monitor (VMM) [69]. The VMM allows the execution of several different VMs, all having different operating systems, on a single host entity. This hypervisor is responsible for providing an abstract interface for the hardware and for partitioning the underlying hardware resources. The underlying resources are then available to the VMs through the VMM, which maintains full control of the resources given to the VM.

The main contribution of this technology is the provision by the VMM of strong isolation between virtual machines. The realization of the processor's ring concept differs depending on the virtualization method [180]. However, common ground for all virtualization methods is that the VMM runs in the highest ring level, typically ring 0, and is, therefore, able to isolate different compartments.

Virtualization can be divided into two main categories [132]: *Paravirtualization* and *Full Virtualization*. The level of abstraction provided by Paravirtualization is different from the level of abstraction provided by the underlying hardware.

In a paravirtualized environment, costly, non-efficiently virtualizable instructions are substituted with efficiently virtualizable instructions. For this, the operating systems running in a paravirtualized environment are modified and non-efficiently virtualizable instructions are substituted. In contrast, full virtualization emulates these costly non-virtualizable commands, through *binary translation* [155, 165]. For this purpose, the VMM, rather than the VM, has to catch, isolate and interpret these instructions and then return control to the VM afterwards. Therefore, the operating system running in a virtual machine does not have to be modified, but can be directly executed. Binary translation is normally linked with an extensive performance overhead, compared to the fast and efficient Paravirtualization approach.

Since the x86-processor architecture is not efficiently virtualizable [129], Intel and AMD have introduced Virtualization Technology support in their processor architecture. In the context of this thesis, the Intel VT-X/I architecture [180, 37, 52] is of special interest, since the approach presented in this thesis uses functionalities provided by this architecture. The Intel VT-X/I architecture augments the x86-processor architecture with two new forms of CPU operation, namely VMX root operation, in which the VMM runs, and VMX non-root operation, in which the guest-systems run. Both operation support the privilege set of the x86 architecture. The VMM is then typically run in ring 0 of the VMX root mode and the guest system is run in ring 0 of the VMX non-root mode. The processor additionally provides a special purpose structure called *virtual machine control structure* (VMCS). In this structure, state information of the virtual machine are stored in and loaded into the processor if a state transition is per-

formed. A state transition to a VM is called `vmentry` and the transition back to the VMM is called `vmexit`.

2.4 Formal Verification of Protocols

Formal verification methods are used to prove that a security protocol is correct and behaves as specified. These techniques are more reliable than informal arguments about the correctness of a protocol. The purpose of this section is to give a very short overview of some approaches that enable protocol verification. Further information about formal verification can, for instance, be found in [22] and [109].

Basically, formal verification methods can be divided into methods based on theorem proving and methods based on model checking. We will shortly describe these two methods in the next two sections.

2.4.1 Theorem Proving

Formal verification methods that are based on theorem proving consider all possible protocol behaviours and allow checking that the protocol satisfies a set of correctness conditions. One advantage of theorem proving is that these methods are more suitable to prove a protocol correct while the major disadvantage is that attacks on the protocols cannot be found. Classical examples of approaches that are based on theorem proving are the BAN logic [27], the GNY logic [71], and Isabelle/HOL [116]. However, theorem prover often suffers from the fact that they require hand written proofs and often the idealization of a protocol. Even worse, is that many examples exist in the literature, where a protocol was proven correct using theorem proving and, nevertheless, a security vulnerability has later been found. One example is the Needham-Schroeder protocol where the security property (that the delivered nonces are not accessible by an adversary) can be proven correct [27]. However, it has been shown that this security property is not satisfied and a security vulnerability emerges [104, 115].

2.4.2 Model Checking

Methods that utilize model checking techniques consider a large, but finite, number of possible protocol behaviours, and allow checking that they satisfy a set of correctness conditions. Methods based on model checking often construct a complete state space of the protocol's behaviour. An exhaustive search is then performed on this state space in order to find a path in the state space that yields in a state where the correctness conditions are violated. In contrast to theorem proving methods, model checking is more appropriate for finding attacks on protocols, while they are rather useless for proving the correctness of a protocol. One interesting property of model checking methods is that they can provide an attack trace when a protocol does not satisfy a correctness condition. However, in contrast to theorem proving methods, they do not yield a symbolic proof which gives insight why a protocol is correct. Example model

checkers are FDR, Mur ϕ [47], SHVT [74], and AVISPA [182]. The latter one is used throughout this thesis to verify the correctness of the developed protocols.

2.4.3 Comparison of selected tools for protocol analysis

In Figure 2.2 we give a short overview of different tools for protocol analysis. We only present some selected and well-known tools and integrate them into their respective category. However, we do not give information about how useful a tool is in providing assurances about protocol security. This is very difficult to answer and clearly out of the scope of this thesis.

<i>Tool</i>	<i>Type</i>	<i>Used Language</i>
FDR [104]	Model checker	CASPER
Mur ϕ [47]	Model checker	Special
Brutus [36]	Model checker	Special
SHVT [74]	Model checker	APA
AVISPA [182]	Model checker	HLSPL
Isabelle [116]	Theorem proving	HOL
BAN [27]	Theorem proving	Special
GNV [71]	Theorem proving	Special

Figure 2.2: Summary of selected tools for protocol analysis

2.5 Notation and Definitions

In this section, we provide the notations and definitions used in this thesis. We will first introduce the cryptographic notation, followed by the definitions.

2.5.1 Cryptographic Notation

In the remainder of this thesis we use the following notation. $E(m, e)$ denotes the *encryption* of data m using an encryption function E and encryption key e . *Encrypted data* m that is encrypted with the key e is denoted with $\{m\}_e$. The *decryption* of $\{m\}_e$ using a decryption function D and the decryption key d is denoted with $D(\{m\}_e, d)$. If the decryption key d is equal to the encryption key e we speak of symmetric cryptography. If the decryption key d is not equal to e , we speak of asymmetric cryptography.

A symmetric key that is shared between the entities A and B is denoted with K_{AB} . When a key is shared between entities, both entities possess the same key to decrypt or encrypt data. The private part of an asymmetric key of entity A is denoted with K_A^{-1} and the public part is denoted with K_A .

The application of a cryptographic *hash function* h to data m is denoted with $h(m)$. A one-way *hash chain* [98] is a sequence of hash values of some fixed length l generated by a hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^l$ by applying the hash function h successively to a seed value c_0 so that $c_{v+1} = h(c_v)$, with $v = 0, 1, \dots, n - 1$.

In the remainder of this thesis, we use the popular *Alice-and-Bob* specification style [108]. A signed message m that is signed with the key K_A^{-1} is denoted as $\{m\}_{K_A^{-1}}$. Before a message is signed, the hash of the message is computed and the signature is computed on this hash value. To verify a signature, both the signed message and the plaintext of the message before the hash function is applied are required. A message of this type is shown in Figure 2.3.

$$A \rightarrow B : m, \{m\}_{K_A^{-1}}$$

Figure 2.3: A transferred signed message including transmittal of the message in plaintext

However, we will not explicitly state that besides transferring a signed message from one entity to another entity, the message is also delivered in plaintext. A protocol of this type is shown in Figure 2.4. In the context of this thesis, the following protocols are thus interchangeable and we will, for simplicity reasons, exclusively use the notation shown in Figure 2.4.

$$A \rightarrow B : \{m\}_{K_A^{-1}}$$

Figure 2.4: A transferred signed message

2.5.2 Definitions

We denote a specific state of a client configuration as *platform configuration*. The platform (i.e., attesting system) that delivers its integrity to another platform using platform attestation is also referred to as prover, since it wants to prove to a verifier that he is in a trusted system configuration.

The terms trust and trustworthy have a wide spectrum of meanings. Both terms are often used interchangeably. However, to make the meaning of these terms exact, we provide the following definitions, which are derived from [153]:

- **Trust level:** The extent to which someone who relies on a system can be confident that the system meets its specifications, i.e., that the system does what it claims to do and does not perform unwanted functions.
- **Trusted:** The term trusted describes a system that operates as expected, according to design and policy.
- **Trustworthy:** A system is trustworthy if it is trusted and the trust can also be guaranteed in some convincing way, such as through a formal analysis or a code review.

In the sequel of this thesis, we only use the term *trusted* since it describes exactly the characteristics of a platform we are considering.

Part II

Attestation Protocols

Chapter 3

Secure Attestation

In this chapter, we show that in order to enable a system to validate the platform integrity of a remote system, a secure attestation channel is required. A secure attestation channel ensures authenticity of platform integrity measurements and establishes a cryptographic channel to the source of platform integrity measurements. This chapter contains material previously published in: *A Robust Integrity Reporting Protocol for Remote Attestation* [164] and *Hades - Hardware Assisted Document Security* [130].

3.1 Introduction

The increasing complexity of IT-systems is a major factor for the growing number of system vulnerabilities. Once a vulnerability is exploitable, malware can use the vulnerability to infect the system, which can result in a system crash, or that a virus or worm spreads over the system and makes sensitive data public. If a software system has been infected by malware, it is also a serious threat to other non-compromised software systems since it could infect other systems while they communicate with each other.

One promising approach to detect a compromised system is to use attestation protocols as specified by the Trusted Computing Group (TCG) [179]. This approach enables to ensure that a system has not been compromised before the communication starts. One important mechanism for realizing attestation protocols is to securely report the integrity of a specific system. This mechanism can be applied in many scenarios and different applications trying to ensure a malware free communication. These scenarios include Enterprise Security [141], E-Commerce [158] or Information Rights Management [137]. Besides counteracting the typical threats posed by malware with attestation protocols, these protocols can also be used to ensure that a specific software application is in a specific pre-defined state. This approach is particular important in the context of Information Rights Management to ensure an untampered and non-manipulated client software.

However, designing attestation protocols that enable secure integrity reporting is a tedious task, since a number of challenges must be solved. In this chapter, we present these challenges and extract requirements that must be satisfied in order to solve the

presented challenges. If all our extracted requirements are satisfied, a *secure attestation channel* can be established which ensures authenticity of integrity measurements and authenticity of the attestation channel. One important requirement is that these protocols must implement a mechanism that secures the protocols against masquerading attacks¹ [133], where an attacker masquerades his platform configuration as being trusted. These attacks are characterized through the fact that an integrity challenge is relayed to another platform that has a trusted platform configuration. The trusted platform answers the challenge, which is transferred back to the requester.

This chapter is organized as follows. We will first present in Section 3.9 other work that is related to the protocols proposed in this chapter. We will then present a classification of attestation techniques in Section 3.2. In Section 3.3 we explain the integrity reporting protocol as proposed by the TCG [173] and highlight a critical attack on this protocol. In Section 3.4 the underlying attack is evaluated and the reasons why the attack works are explained. Section 3.5 shows that this type of attack can easily be avoided by integrating a key-establishment into the protocol and thus establishing a secure attestation channel. We will then perform a security analysis of our proposed protocol in Section 3.6. In Section 3.7 we will show how a secure attestation channel can be enhanced and integrated inside the TLS-protocol, ensuring integrity reporting and mutual-authentication. The security of our proposed enhanced TLS-protocol is then analyzed in Section 3.8. Finally, we conclude in Section 3.10.

3.2 Classification of Attestation Techniques

In this section, we compare different attestation techniques and perform a classification. We classify existing approaches based on the underlying attestation concept. Based on this classification, we show how the relevant approaches realize attestation techniques.

3.2.1 TPM-Based Binary Attestation

Many attestation protocols (e.g., [164, 152, 103, 137]) are based on the TPM's ability of obtaining measurements of platform characteristics that describe the integrity of a platform. These approaches are mainly developed for non-resource constrained computer systems and require each communication partner to perform public key cryptography. The complete system configuration, as denoted in the PCRs of the attesting entity, must be transmitted to the verifying entity. The verifying entity explicitly evaluates the trust level of the attested entity by comparing the received SML and PCR values with given reference values. Since the verifying entity receives the current platform configuration directly, we refer to this as *explicit attestation*. The exact mechanisms of performing an explicit attestation can vary and do not necessarily require using the `TPM_Quote` command.

The sealing concept of the TPM enables another means of attestation without directly transferring the platform configuration (PCR values and SML). We refer to this

¹These attacks are in [70] referred to as relay attacks.

as *implicit attestation*. These approaches are based on the sealing functionality of the TPM. Sealing provides a means to bind data to a certain platform configuration. The TPM releases, i.e., decrypts, this data only if the current platform configuration is identical to the initial configuration. Publications that include some sort of *implicit attestation* include [96] and [159]. This approach is able to minimize the amount of transmitted data and does not necessarily require public key cryptography on resource constrained systems. How this can be achieved is discussed in Chapter 6.

The disadvantage of this approach is that software updates change the values inside the PCRs. Since this results in inaccessible sealed data, this approach is not applicable in non-resource constrained computer systems, where the software configuration often changes through legitimate system updates.

Attestation techniques that are based on the TPM-based binary attestation require some sort of measurement architecture. This architecture is responsible for measuring every software component before execution using the `TPM_Extend` command, which includes the measurements into the platform configuration registers.

Examples for such an architecture are the Integrity Measurement Architecture (IMA) developed by IBM [142] or the Bear/Enforcer project [106]. IMA provides an extension to the Linux-Kernel that performs this task and the Bear/Enforcer project includes a TPM-enabled Linux Security Module (LSM) to compare hash values of applications with reference values.

3.2.2 Software-Based Attestation

The main disadvantage of the TPM-based binary attestation is that the platform configuration only reflects the initial load-time configuration. Therefore, memory modifications during the runtime, e.g., buffer-overflows, cannot be detected. To overcome this shortcoming, an attestation software may measure randomly selected parts of the memory and report the obtained measurements to a remote party. In this case, the attestation software forms the trust anchor which must be protected against tampering.

In [150, 149, 147], approaches that are based on measuring the execution time of an attestation routine are introduced. A verifier specifies randomly chosen memory regions and transfers these randomly chosen regions as challenges to the prover. Based on these challenges, the prover uses his own attestation process to calculate the corresponding hash values of his own memory. The computed value is then transferred back to the verifier who can compare it to trusted system states. Malware which is loaded into the memory would result in a modified memory which can be detected by the verifier. The main idea of these approaches is that the attestation routine of the prover cannot be optimized further, i.e., the execution time cannot be made faster, which prevents an adversary from injecting malicious code without detection. For example, it would be necessary for an adversary who injected malicious code into the memory to also modify the attestation routine so that the attestation routine does not measure the injected malicious code. However, modifying the attestation routine, e.g., by adding additional *if-statements* that jump to an alternative memory region if a malware infected memory region is measured, would result in an increased execution time of the attestation

routine and can, thus, be detected. One disadvantage of this approach is that the success of this scheme relies critically on the optimality of the attestation routine and on minimal time fluctuations of the expected responses.

However, in many scenarios (e.g., wireless sensor networks or distributed systems with external influences) time intervals for responses can vary. In these cases the attestation would fail, even though a component is in a trusted system state. In scenarios where attestation along multiple hops is required or external interferences prevent an exact time measurement, timing-based software attestation techniques are not applicable.

Other works that try to adapt software-based attestation include [114] and [30]. However, the main disadvantage of these approaches are that they are highly vulnerable to masquerading attacks, as we will see in Section 3.3.3.

3.2.3 Property-Based Attestation and Semantic Attestation

In contrast to the TPM-based binary attestation, a number of proposals have been made which focus on semantic attestation based on attesting the behavior of software components (e.g., [75, 139, 31]). The idea behind these approaches is that a platform should not depend on specific software configurations, but on properties that a platform offers. A webbrowser of two different vendors would, thus, have the same properties. However, the authors do not specify how a property exactly looks and how a property is generated. In addition, semantic attestation also requires a TCG-enhanced boot process to ensure that a small operating system kernel applies mechanisms that enforce the semantic attestation.

3.3 Masquerading Attacks on Attestation protocols

In the following section, we will mainly focus on TPM-based binary attestation. These techniques can be used to provide assurances that a remote platform or the own platform is in a particular state. This especially becomes relevant in systems for Information Rights Management, where a client enforces a certain policy to prohibit unauthorized use, copy or redistribution of intellectual property [125].

In this context, the policy enforcement mechanism is executed on the client-side, so an attacker can deactivate this mechanism by modifying the local client software. To avoid these kinds of attacks, one may attest the status of the client platform using remote attestation before sending sensitive data to the client. Thus, the sender has a confirmation about the trust level of the platform in question.

However, as we will see in the following, the protocols specified by the TCG have a vulnerability which allows an attacker to masquerade an untrusted platform configuration as trusted. An attacker who controls one malicious and one honest client may bypass the remote attestation by spoofing his malicious client to be the honest one. The attacker simply relays all attestation queries and sends them to the client which is in a trusted system state. The honest client sends the answer back, which must only be transferred back to the requester.

3.3.1 Integrity Reporting Protocols

In this section, we discuss an integrity reporting protocol proposed by [142] that is consistent to the process of remote attestation as specified in [177]. We will show that the protocol is vulnerable to a masquerading attack with the result that the attacker is able to masquerade his own platform configuration. Protocol 3.3.2 illustrates the remote attestation of B against A and provides the background information on integrity reporting protocols.

Protocol 3.3.2: Integrity reporting protocol proposed by Sailer et al. [142]

SUMMARY: B answers the attestation challenge of platform A

RESULT: Integrity reporting

1. *System setup.*

A must acquire (and validate) the certificate of the Privacy-CA to validate $Cert(AIK, K_{AIK})$

2. *Protocol messages.*

$$A \rightarrow B : Na \quad (1)$$

$$A \leftarrow B : SML, Cert(AIK, K_{AIK}), \{Na, PCR\}_{K_{AIK}^{-1}} \quad (2)$$

3. *Protocol actions.*

- (a) A chooses a non-predictable nonce Na and delivers it to B (1).
- (b) B loads the AIK from the protected storage of the TPM by using the storage root key (SRK). In the next step, B performs a `TPM_Quote` command, which is used to sign the selected PCRs and the provided nonce with the private key K_{AIK}^{-1} . Additionally, the prover retrieves the stored measurement log (SML).
- (c) B sends in message (2) the response consisting of the signed *Quote*, signed nonce and the SML to A . B also delivers the AIK credential ($Cert(AIK, K_{AIK})$) which consists of the AIK that was signed by a Privacy-CA.
- (d) A validates if the AIK credential was signed by a trusted Privacy-CA thus belonging to a genuine TPM. A also verifies whether $Cert(AIK, K_{AIK})$ is still valid by checking the certificate revocation list of the trusted issuing party. This step was also thought to detect masquerading attacks by comparing the unique identification of B with the system identification given in $Cert(AIK, K_{AIK})$. Nevertheless, this verification does not discover masquerading attacks as we will see in the next section.

- (e) Finally, A verifies the signature of the $\{Na, PCR\}_{K_{AIK}^{-1}}$ and checks the freshness of Na . Based on the received stored measurement log and the PCR values, A processes the SML and re-computes the received PCR values. If the computed values match the signed aggregate, the SML is considered valid and untampered. A now only verifies if the delivered integrity reporting values match given reference values, thus A can decide if the remote party is in a trusted system state.

Protocol 3.3.2 is based on a challenge-response authentication involving the TPM. This process ensures freshness and authenticity of integrity information and prevents that old integrity information or non-authentic integrity information are replayed in a new protocol run. This requirement is necessary to reach the protection goal of *authenticity* and *integrity* of delivered data over an attestation channel. We summarize this in the following requirement for a secure attestation channel:

Requirement 1: Attestation techniques must always ensure freshness and authenticity of integrity information.

All attestation protocols that are consistent with the TCG-defined integrity reporting satisfy Requirement 1. They are resistant against replay attacks and ensure authenticity of integrity information. However, satisfying Requirement 1 is not sufficient to enable secure integrity reporting, since Requirement 1 does not require a mechanism to detect or prevent masquerading attacks. In the context of attestation, masquerading attacks are a special form of attack and cannot be prevented with classical approaches that ensure the authenticity of a message. We will discuss why this is critical in the following section.

3.3.3 Masquerading Attacks on Attestation Protocols

In this section, we present a masquerading attack on Protocol 3.3.2. This shows that an attestation protocol which only satisfies Requirement 1 cannot be used for secure integrity reporting. The attacker considered here has two platforms under his control. One platform runs a trusted operating system with the client software that enforces a certain policy, e.g., an IRM-client. This platform (C) runs the original client software and is therefore untampered. C is also equipped with a genuine TPM that supports the policy enforcement of the IRM-client. The attacker is also in control of one malicious client platform (M), where he wants to gain control of protected digital content. We refer to this client as malicious client, since its enforcement mechanism has been tampered with and it is not conform to the original client software. We require for our attack that C answers the request from M , i.e., C is not configured to answer only requests from A . This is a necessary requirement for the success of the attack and is discussed in detail in Section 3.4.

In our attack, the attacker bypasses the remote attestation of M , by using the platform configuration of the honest client to attest his malicious client running on M . For the sake of simplicity, we only show the transferred messages, since the actions taken by A and B are analogous to the full protocol shown in Protocol 3.3.2. The attack presented here is some special type of the *Grandmaster Chess Problem* or *Mafia Fraud*, which has already been discussed in the literature [21, 49, 3].

$$A \rightarrow M : Na \quad (1)$$

$$M \rightarrow C : Na \quad (2)$$

$$M \leftarrow C : \text{SML}, \text{Cert}(\text{AIK}, K_{\text{AIK}}), \{Na, \text{PCR}\}_{K_{\text{AIK}}^{-1}} \quad (3)$$

$$A \leftarrow M : \text{SML}, \text{Cert}(\text{AIK}, K_{\text{AIK}}), \{Na, \text{PCR}\}_{K_{\text{AIK}}^{-1}} \quad (4)$$

Figure 3.1: Masquerading attack on the integrity reporting protocol

Figure 3.1 depicts the attack against the integrity reporting protocol. The challenging party A wants to securely validate the integrity of the attesting malicious system M using Protocol 3.3.2. The malicious system M itself transfers all messages from A to the honest client C . As a result, the integrity information is still authentic while the authenticity of the attestation channel is violated.

This masquerading attack is also relevant in software-based attestation protocols. All proposals [146, 150, 149, 147, 114, 30] are vulnerable to this attack. In order to present this described attacks, some proposals (e.g., [146] and [150]) assume that they have an authenticated and monitored communication channel, which inhibits the presented attack. However, we believe that this assumption is too restrictive since it only makes software-based approaches applicable in a very small set of scenarios.

3.3.4 Attestation over Secure Channels

Transferring the SML in plain-text (and thus the whole platform configuration) to a remote party raises severe privacy concerns. The remote party and every party that is able to eavesdrop on the communication channel is able to discover the full platform configuration, including all running processes, by simply examining the SML. This information could be used to collect extensive information about one platform. It was therefore intended to integrate integrity reporting protocols inside another cryptographic channel, and refrain from performing an explicit attestation over insecure channels [176]. This requirement is necessary to reach the protection goal of *confidentiality* of the attestation channel and is summarized in Requirement 2:

Requirement 2: The explicit attestation must be performed over secure channels.

However, even if the protocol specified by the TCG is integrated into a TLS channel, the protocol is not robust against the relay attack shown in Figure 3.1. To show that the

above discussed attack cannot be prevented if another cryptographic channel is used, we integrate the IRP in the TLS-protocol after a symmetric key has been established.

Figure 3.2 shows the simplified attack on a TLS-secured integrity reporting. The figure shows the challenger (platform A) that wants to attest the client (M) before protected data is transferred and M obtains access to a particular service offered by A . Platform A and platform M first establish a mutually authenticated TLS channel. After establishing this secure channel, A delivers the attestation challenge to platform M . M receives the attestation challenge, establishes a secure channel with platform C and masks the received attestation challenge of A as fresh attestation challenge for platform C . C computes the attestation response message using its TPM and delivers the response to platform M . M receives the response and answers the attestation challenge of platform M . Thus, the platform of M is authenticated on the basis of the provided information through platform C .



Figure 3.2: Masquerading attack on the TLS-secured remote attestation

To further analyze why this attack is successful even if a secure cryptographic channel is used, we look at the TLS-protocol in detail and analyze which messages are involved in establishing the TLS channel. To enable mutual authentication of both entities, we use the `RSA_DHE` TLS-mode. We abstract the TLS-protocol in order to not provide unnecessary details and to reduce the complexity of the protocol. The presented abstracted version of the TLS-protocol is an enhanced version of the one presented in [117]. Protocol 3.3.5 shows the protocol steps. Here A acts as server, M is both client and server, and B is the prover.

Protocol 3.3.5: Integrating TLS and IRP

SUMMARY: M answers the attestation challenge with the help of platform B
 RESULT: Mutual authentication, key establishment, relayed integrity reporting

1. Notation.

- Sid denotes the TLS security parameters and the protocol ID
- $Cert(X, K_X)$ is the public key certificate of X
- Px denotes the cryptographic engines which are supported by X
- DH_X denotes the public Diffie-Helman key of entity X
- K_{AM} denotes the negotiated key between A and M

K_{CM} denotes the negotiated key between C and M

2. *System setup.*

A must acquire (and validate) the certificate of the Privacy-CA to validate $Cert(AIK, K_{AIK})$.

All entities must possess the certificate of the certification authority, in order to validate the received certificates.

3. *Protocol messages.*

$A \leftarrow M$:	M, Nm, Sid, Pm	(client hello)	(1)
$A \rightarrow M$:	A, Na, Sid, Pa	(server hello)	(2)
$A \rightarrow M$:	$Cert(A, K_A)$	(server certificate)	(3)
$A \rightarrow M$:	$DH_A, \{DH_A, Na, Nm\}_{K_A^{-1}}$	(server key exchange)	(4)
$A \leftarrow M$:	$Cert(M, K_M)$	(client certificate)	(5)
$A \leftarrow M$:	DH_M	(client key exchange)	(6)
$A \leftarrow M$:	$\{DH_A, DH_M, Na, Nm\}_{K_M^{-1}}$	(certificate verify)	(7)
$A \rightarrow M$:	$\{Finished\}_{K_{AM}}$	(client finished)	(8)
$A \leftarrow M$:	$\{Finished\}_{K_{AM}}$	(server finished)	(9)
$A \rightarrow M$:	$\{Ni\}_{K_{AM}}$	(attestation challenge)	(10)
$C \rightarrow M$:	C, Nb, Sid, Pb	(client hello)	(11)
$C \leftarrow M$:	M, Nm, Sid, Pm	(server hello)	(12)
$C \leftarrow M$:	$Cert(M, K_M)$	(server certificate)	(13)
$C \leftarrow M$:	$DH_M, \{DH_M, Nb, Nm\}_{K_M^{-1}}$	(server key exchange)	(14)
$C \rightarrow M$:	$Cert(C, K_C)$	(client certificate)	(15)
$C \rightarrow M$:	DH_C	(client key exchange)	(16)
$C \rightarrow M$:	$\{DH_M, DH_C, Nm, Nb\}_{K_C^{-1}}$	(certificate verify)	(17)
$C \rightarrow M$:	$\{Finished\}_{K_{CM}}$	(client finished)	(18)
$C \leftarrow M$:	$\{Finished\}_{K_{CM}}$	(server finished)	(19)
$C \leftarrow M$:	$\{Ni\}_{K_{CM}}$	(attestation challenge)	(20)
$C \rightarrow M$:	$\{Cert(AIK, K_{AIK}), SML, \{PCR, Ni\}_{K_{AIK}^{-1}}\}_{K_{CM}}$	(attestation reply)	(21)
$A \leftarrow M$:	$\{Cert(AIK, K_{AIK}), SML, \{PCR, Ni\}_{K_{AIK}^{-1}}\}_{K_{AM}}$	(attestation reply)	(22)

In the first nine steps, a secure TLS channel is established between A and M . Both A and M are in possession of certificates to authenticate themselves. They use the TLS-

mode `DHE_RSA` and both generate and exchange ephemeral Diffie-Helman keys DH_M and DH_A (steps 4-7), which are used to compute the pre-master-secret. The negotiated pre-master secret is then combined with the exchanged nonces Nb and Na and converted into the master-secret (Cf. [45] or [54] for more details). The negotiated shared key K_{BM} is then derived from the master-secret. The TLS-protocol finishes with transmitting *Finished* messages. *Finished* messages comprise all messages that both parties have been delivered or received until now in the whole protocol run. These messages ensure that the key exchange and authentication process of the preceding steps were successful and that no man-in-the-middle has altered the transferred messages.

After the TLS-protocol run, A delivers the attestation challenge message to M which consists of the nonce Ni (step 10). M takes then the position as server and another TLS channel is established between M and C (steps 11-19). The computed shared key K_{BM} is then used to masquerade the nonce Ni as a fresh attestation challenge and delivers it to C . C will answer with the attestation reply message, which is decrypted by M and then delivered to A , re-encrypted with key K_{AM} .

3.3.6 Analysis of the Attack

Protocol 3.3.5 shows that the attack is still possible even if a secure channel is used. M can successful masquerade his own platform configuration, as the message (22) gives no evidence that this message was generated on the specific machine that is under control of M . This uncertainty is caused by the independency of the private key that corresponds to $Cert(M, K_M)$ (message 5) and the private key that corresponds to $Cert(AIK, K_{AIK})$. Both certificates are issued by independent certification authorities. One authority is responsible for issuing certificates for a certain user or a machine and the other is responsible for issuing *AIK certificates*. Exacerbating this problem is the fact that an *AIK* is specific for a TPM and thus clearly identifies a unique machine. But in contrast to that, the TLS certificate is not necessarily specific for a certain machine.

To overcome that problem Goldman et al. [70] introduce a platform property certificate that links an *AIK* certificate to a TLS certificate. Unfortunately, that approach is not able to prevent an attack if M and C collaborate. In that case, the TLS certificate and the corresponding private keys of C are transferred to M , i.e., M is later authenticated on the basis of $Cert(C, K_C)$. Since standard TLS applications, e.g., web browsers, support exporting the keys by the platform owner, this attack is a very realistic attack; the attacker can transfer the certificates and the private keys to the non-conformant host. It should also be noted that even if other devices that do not support exporting private keys, such as smart cards, are being used for establishing the TLS channel this attack cannot be detected. This is caused by the fact that if an attacker is the legitimate owner of the device, he can use it twice to authenticate himself and thus establish two different secret channels.

The shortcoming of the integrity reporting protocol is caused by the restricted usage possibilities of *AIKs*. According to [179] the *AIKs* can exclusively be used for proving the authenticity of a platform. This means that *AIKs* can neither be directly used to establish secure channels nor to authenticate communication partners. Therefore, the

challenger cannot be sure whether the received message belongs to the attesting system. He only knows that he received a message from a genuine TPM. Additionally, the possession of multiple *AIKs* is possible, the only requirement is that the corresponding certificate must be certified by a trusted Privacy-CA or via DAA thus belonging to a valid Endorsement Key. As a consequence, the challenging party cannot map *AIKs* to users and thus to a specific cryptographic session.

Even if the server is in possession of $\text{Cert}(\text{AIK}, K_{\text{AIK}})$ and forbids the creation of new *AIKs*, masquerading attacks cannot be prevented, because the attesting system *M* pretends that it is in possession of the corresponding K_{AIK}^{-1} by using the integrity values with a valid signature from platform *B* (Step 6 in Figure 3.2). The challenging system *A* is therefore not able to detect this attack in step 7 (Figure 3.1), since the attesting system delivers all information that identifies it as platform *B*. After the platform is authorized, the malicious platform is assumed to be trusted.

3.4 Evaluation of Possible Solutions

In this section, we present some approaches that change the overall system architecture to alleviate the problem. However, we show for each of these solutions why they fail to prevent the described attack.

3.4.1 Verifying Attestation Challenges

Since the attacker cannot modify the software on the honest platform, the honest platform may decide which attestation challenge (represented by a nonce) it will answer and which it will reject. For this purpose, the challenger could sign the attestation challenge and the prover validates the signature using a pre-installed certificate before he answers the challenge. In this case, the server's certificate must be included in the integrity measurement of the client software to detect a manipulation on that certificate.

However, any arbitrary software component that is able to answer attestation challenges can respond to the malicious challenge. Thus, an attacker can still use such a software to circumvent the verification of the challenge of the client software. Nevertheless, the verifying party detects that this software component is listed in the SML, where the correct software is also listed. On the downside, the SML does not tell which software issued the answer to the attestation challenge. The attacker can either use a software supplied by himself to answer the attestation challenge or he can misuse an existing application that serves a legitimate purpose. In the latter case, the issuing software appears to be trusted, because its software vendor might offer valid SML-reference values.

Regardless of which client software responds to the attestation challenge, the only requirement is that the nonce, which was sent from the server-application (platform *A*) is used for the attestation. Hence, bugs or design flaws that cause that any arbitrary attestation challenge are answered, affect other trusted client software. One approach is to forbid the installation of other client software components. However, this leads to high restrictions which can hardly be accepted in open environments.

3.4.2 Using Shared Secrets

One method may be to exchange a shared secret key between server and client, which is generated in the TPM. This shared secret key must be marked as a non-migratable key and the public part must be transferred to the server. This key is used to securely exchange a key for the upcoming communication. However, this public key must be transferred through a protected second channel, otherwise the server is not sure that the key belongs to the correct communication partner.

3.4.3 Using *AIK* as Session Key

Another method is to use the public part of the *AIK* as encryption key to exchange the shared key for the upcoming communication. In this case, the upcoming traffic cannot be decrypted by the attacker, since he does not have the private portion of the *AIK*, which is stored in the protected storage of the TPM. This approach is for example proposed in [97] to ensure that transferred data is directly bound to a specific *AIK*. However, actual TPMs do not allow to decrypt data using the private part of the *AIK*. This is due to the specification that clearly forbids using the *AIK* as encryption key ([179], pp. 57).

3.4.4 Using Network Monitors

One alternative solution to achieve a binding between the platform that issues the measurements and the challenger is to use a network monitoring agent [34]. This agent monitors all incoming and outgoing network connections and integrates a measurement of the network events into the PCRs. Thus, the challenger can determine if the network address of the platform that provided the measurements is equal to the network address of the challenger. At first glance, this approach seems to be very elegant, however, it suffers from the fact that it is not usable if the system is behind a router that uses network address translation (NAT). In addition, an attacker can easily modify his own network address, since network addresses do not provide any means of authenticity. It is, thus, very likely that the attacker is able to spoof the network address and is able to impersonate the address of the platform that provided the measurements.

3.4.5 Conclusion

To summarize this section, all discussed solutions are not able to prevent the described attacks. As a result, new approaches are required that prevent the masquerading attack.

3.5 Secure Attestation Channels

In this section, we show how a secure attestation channel can be established. A secure attestation channel ensures authenticity of platform integrity measurements and establishes a cryptographic channel to the source of platform integrity measurements.

3.5.1 Preventing Masquerading Attacks

The attack presented in Protocol 3.3.3 has similarities to the *Mafia Fraud* or the *Chess Grandmaster Problem* on identification protocols and some solutions have already been proposed in the literature [21, 49, 3]. These solutions try to detect whether a potential adversary is trying to impersonate as an authentic entity and either rely on exact time measurements [21] or on using more than one communication channel [3]. However, the goals in the context of preventing *Mafia Fraud* on identification protocols are not identical to the goals in preventing masquerading attacks on integrity reporting. In the latter case it is sufficient that a potential malicious host, who is relaying messages and placed between server and the collaborative host, is excluded from the following communication flow. In the context of integrity reporting, it is sufficient, that the endpoint of the communication is extended to the TPM. For this purpose, a cryptographic channel must be established, that ensures that the upcoming communication is bound to the currently attested system and is, thus, authentic. Thus, in order to achieve *authenticity* of the attestation channel, a third requirement must be satisfied. We summarize this in the following Requirement 3:

Requirement 3: Attestation protocols must integrate a key-establishment component that ensures that the established attestation channel is authentic.

To protect the integrity reporting protocol against masquerading attacks, we enhance it with a key agreement protocol. The modified integrity reporting protocol is shown in Protocol 3.5.2 with the extension to use Diffie-Hellman parameters. The depicted protocol only shows the transferred messages, since the TPM operations are analogue to Protocol 3.3.2. We implement this key establishment by injecting additionally information necessary to negotiate a key into the *AIK* signed attestation challenge.

The specification of the `TPM.Quote` command allows us to include additional data and to sign it with the *AIK*. Thereby, it is possible to use the challenge-response messages for the key-exchange. One possibility is to adopt the Diffie-Hellman (DH) key exchange protocol [46] and to integrate it into the integrity reporting protocol. Alternatively, RSA [128] can be used and generate the corresponding keys inside the TPM. Subsequently, these keys are used to exchange a shared session key which is used to encrypt the upcoming communication. Since the TPM does only support internal symmetric encryption, the CPU must be used for that purpose. The main difference between both approaches is that the DH parameters must be generated by the CPU whilst the RSA keys are generated by the TPM. Both approaches are equivalent in terms of security, since they both offer a secure way to exchange a key. Computing the RSA key with the TPM offers no advantage compared to computing the DH parameters with the CPU, since the shared communication key must be passed to the CPU anyway. In both cases the required random numbers can be computed by the secure random number generator of the TPM.

Protocol 3.5.2: Robust Integrity Reporting Protocol

SUMMARY: B answers the attestation challenge of platform A

RESULT: Key establishment with key confirmation: integrity reporting

1. *Notation.*

K_{AB} denotes the negotiated key between A and B

2. *System setup.*

A must possess the certificate of the Privacy-CA to validate $Cert(AIK, K_{AIK})$

3. *Protocol messages.*

$$A \rightarrow B : Na, g^a \bmod p, g, p \quad (1)$$

$$A \leftarrow B : Cert(AIK, K_{AIK}), SML, \{Na, PCR, g^b \bmod p\}_{K_{AIK}^{-1}} \quad (2)$$

$$A \rightarrow B : \{Nb\}_{K_{AB}} \quad (3)$$

$$A \leftarrow B : Nb \quad (4)$$

4. *Protocol actions.*

- (a) *Precomputation by A.* A selects an appropriate prime p and generator g of \mathbb{Z}_p^* ($2 \leq g \leq p - 2$).
- (b) A chooses a random secret a , $2 \leq a \leq p - 2$, and computes $g^a \bmod p$.
- (c) *Attestation challenge.* A sends message (1) to B .
- (d) B also chooses a random secret b , $2 \leq b \leq p - 2$, and computes $g^b \bmod p$.
- (e) B computes the attestation response message by signing the own public key, Na and a set of PCRs using an AIK .
- (f) *Attestation response.* B delivers the signed message together with the SML and the AIK credential to A (2). Finally, B computes $K_{AB} = (g^a)^b \bmod p$.
- (g) A verifies whether the delivered credentials are authentic and verifies if all signatures are valid. Finally, A verifies whether the platform configuration of A is trusted using the SML.
- (h) A also receives $g^b \bmod p$ and computes $K_{AB} = (g^b)^a \bmod p$.
- (i) A sends an encrypted nonce in message (3) to B .
- (j) B decrypts the nonce with K_{AB} and delivers message (4) to A .

The major enhancement is that the prover generates $g^b \bmod p$ and includes $g^b \bmod p$ into its signed attestation response. The public key of platform B is generated using

the client's CPU and injected together with the supplied nonce N_a as *external data* into the `TPM_Quote` operation. Since the `TPM_Quote` command only allows to include 160bits of external data, the package must be reduced to fit the 160bit length by applying *SHA1*. The public key is also transmitted to the challenger in plaintext to enable the challenger to validate the *AIK* signed message. Because B is running a trusted OS with a trusted platform configuration the secret part of the key is not accessible to a potential malicious client. Finally, both parties compute the shared session key K_{AB} and verify its integrity through a second challenge-response protocol.

3.5.3 Alternative Solution

The TPM specification also provides another means to verify that a specific key is bound to a platform configuration. This process requires the use of the `TPM_CertifyKey` operation to prove that a specific key is held in a trusted TPM and bound to a specific platform configuration. This asymmetric key is then used to exchange a shared symmetric key between both parties. This approach also establishes a second cryptographic channel and ensures that the TPM is the endpoint of the communication. In Chapter 5 we will discuss the detailed protocol and show how integrity reporting can be achieved with this process.

This approach also prevents the attack shown in Protocol 3.3.3, however our proposed solution has a better performance. That is caused by the fact, that each time the platform configuration changes, a new TPM key needs to be generated, which is bound to the current platform configuration. This approach is therefore inflexible in particular if the platform configuration changes often over time. Since the generation of a new TPM bound key takes up to 100 seconds [138], this approach is not practicable in every scenario. In addition, this approach would require a new mechanism which is able to reliably detect when the platform configuration changes and a new key needs to be generated. However, it is unclear how such a mechanism could be realized.

3.6 Security Analysis

In this section, we will first perform an informal security analysis of our presented protocol by looking at typical attacks on cryptographic protocols. We will then perform a formal analysis using the AVISPA protocol prover [182].

3.6.1 Informal Security Analysis

This section discusses the presented attestation protocol by looking at typical potential attacks. This includes a man-in-the-middle attack and a version roll back attack.

Man-in-the-Middle Attack

The presented protocol prevents an attacker from spoofing his malicious software configuration since all following messages are encrypted with the computed session key. It

is impossible for M to perform some sort of man-in-the-middle attack and to establish two different cryptographic sessions between the prover and the challenger. This is caused by the fact that he is not able to modify the attestation response (message 2) of B . Remember that his software is in a compromised state and his TPM would provide malicious platform configurations to A . As a consequence, the malicious host has no access to the private part of the generated session key which is stored on platform C . The generated symmetric session key K_{AB} has to be used to encrypt all following data, which can only be decrypted on the machine which possesses the session key. This session key cannot be transferred to the malicious host by the platform owner, since the key is protected by the operating system under normal runtime conditions. Changing these runtime conditions would lead to a non-conformant system state which would have been detected in the attestation phase.

It may be possible for the malicious client M to substitute $g^a \bmod p$ with his own public key $g^m \bmod p$ and transferring this key to B . Since M cannot modify the response from B , A generates the shared session key K_{AB} which is later used for the encryption. The second challenge-response authentication detects this substitution since neither M nor B are in possession of the correct K_{AB} and are therefore not able to encrypt the delivered nonce Nb . It may also be possible that B modifies the common generator g , or the common group p . However, the substitution of $g^a \bmod p$ and the modification of g or p do not lead to a security problem, since in both cases the attacker would not be able to perform the second challenge-response authentication.

Version Rollback Attack

The presented protocol is not compatible with a verifier that expects the insecure existing remote attestation defined by the TCG. This attack was misleadingly classified in [68] as a man-in-the-middle attack. However, this classification is insufficient, since the attack requires that one entity executes the insecure integrity reporting protocol specified by the TCG in [173].

In this attack scenario, three different parties, a verifier, a prover and an adversary, are involved. The adversary tries to relay the attestation challenge of the verifier to the prover, in order to masquerade a trusted system configuration. The verifier and the adversary run an authentication enhanced attestation protocol (e.g., Protocol 3.5.2), while the prover runs the TCG-defined attestation protocol. This attack can be classified as a version rollback attack, in which the adversary masks his public key together with the nonce provided by the verifier as new nonce for the prover. The prover does not verify the syntax of the attestation challenge. He directly signs the masqueraded nonce including public key and delivers the computed signature to the adversary. The adversary simply forwards the obtained message and both adversary and verifier compute the shared key based on the exchanged public keys. However, since the protocol's software integrity of the prover is also reflected in the platform configuration, the verifier will determine the platform configuration as not being trusted. The version rollback attack is therefore only a theoretical attack and fails, since it is not possible to successfully masquerade a trusted system configuration.

3.6.2 Formal Security Analysis

To formally analyze the proposed protocol, we use the AVISPA protocol prover [182]. AVISPA provides a High Level Protocol Specification Language (HLPSL) [10, 33] for describing security protocols and specifying their intended security properties. Protocols specified in HLPSL are then translated into an *Intermediate Format (IF)*. This intermediate format can then directly be used as input for different model checkers supplied by AVISPA (see the AVISPA website for more details²).

AVISPA is a powerful tool to formally analyze cryptographic protocols. However, all tools which can be used to formally analyze cryptographic protocols have in common that they only implement the Dolev-Yao attacker model and are, thus, not capable of detecting special types of insider attacks. Insider attacks are a very powerful attack type, since these attackers can also create authentic messages. The relay attack presented in this chapter is some type of inside attack since the adversary controls a second machine including the private TPM key. Therefore, present cryptographic protocol analyzers cannot detect protocol weaknesses against inside attackers. However, since our proposed protocols also include an authentication phase and a key-establishment, it is verifiable whether both authentication phase and key-establishment are secure against Dolev-Yao attackers [50].

In the following, we present the different roles specified in HLPSL of our attestation protocol. We will first present the challenging entity (Alice), followed by the prover (Bob), the attacker model and the intended security properties.

State Transitions

For specifying the challenger, we will first look at a fragment of the challenger's role. In this fragment, the variables and the challenger's knowledge are specified.

```

role challenger (A, B: agent,
                Ka, Ks: public_key,
                H: function,
                G: text,
                SND, RCV: channel (dy))
  played_by A def=

    local State : nat,
      Na, Nb: text,
      X:text,
      AIK: public_key,
      PCR:text,
      SK:message,
      KEr:message

```

²www.avispa-project.org

The prover's role is specified analogously.

```

role attestor(A, B: agent,
              AIK: public_key,
              Ks : public_key,
              H: function,
              G: text,
              SND, RCV: channel (dy))
  played_by B def=

```

The protocol flow is specified through state transitions. The initial state is set to state 0 and the challenger generates a random nonce (Na) and the secret part of his Diffie-Helman key X' . HLPSTL provides for this process the *new()* command. The challenger then puts on his SND line the generated nonce Na' and the public Diffie-Helman key. The exponentiation to compute a public Diffie-Helman key is specified in HLPSTL through the function *exp*(G, X'), which stands for $G^{X'} \bmod n$. After the package is put on the SND line, the challenger's state is set to 2.

```

init State := 0

transition

0.  State = 0 /\ RCV(start) =|>
    State' := 2 /\ Na' := new()
    /\ X' := new()
    /\ SND(Na'.exp(G,X'))

```

The prover receives the sent package in state one and generates his own secret part of his Diffie-Helman key by assigning $Y' := new()$. He also obtains the current PCR values, which is abstracted, since HLPSTL does not provide any means of directly modeling TPMs. The generated public key is then hashed together with the nonce and the PCRs and signed with the *AIK*. After sending the attestation response, the prover also computes the secret key SK and witnesses the challenger that he is able to compute a valid signature on the challenge.

```

init State := 1

transition

1.  State = 1 /\ RCV(Na'.KEi') =|>
    State' := 3 /\ Y' := new()
    /\ PCR' := new()
    /\ SND({B.AIK}_inv(Ks).{Na'.exp(G,Y')}.PCR')_inv(AIK))
    /\ SK' := H(exp(KEi,Y'))
    /\ witness(B,A,challenger_attestor_na,Na')

```


The challenger receives the attestation response in state 2 and computes the shared secret SK' . He then generates a new nonce Nb' , encrypts the nonce and delivers the encrypted package to the prover.

```
2. State = 2 /\ RCV({B.AIK'}_inv(Ks).{Na.KEr'.PCR}_inv(aik)) =|>
   State' := 4 /\ SK' := H(exp(KEr',X))
   /\ Nb' := new()
   /\ SND({Nb'}_SK')
```

The prover receives the encrypted package, decrypts the nonce with the negotiated key SK' and sends the decrypted nonce Nb' to the challenger. Sending this nonce in plaintext does not result in a security problem since this message only acknowledges correct receipt of the previous message.

```
3. State = 3 /\ RCV({Nb'}_SK) =|>
   State' := 5 /\ SND(Nb')
   /\ witness(B,A,sk1,SK)
   /\ witness(B,A,challenger_attestor_nb,Nb')
```

The challenger receives the answer in state 4 and checks whether the specified security properties are satisfied.

```
4. State = 4 /\ RCV(Nb') =|>
   State' := 6 /\ secret(SK,sk1,{A,B})
   /\ request(A,B,challenger_attestor_na,Na)
   /\ request(A,B,challenger_attestor_nb,Nb')
   /\ request(A,B,sk1,SK)
```

Intruder Model and Environment

We use the Dolev-Yao intruder model [50] to model the attacker and the environment. In this intruder model the attacker is a legitimate user of the network and has full control over all messages that are sent over the network. The attacker can therefore intercept, analyze or modify messages, as well as compose new messages and send the messages to whoever he wants. The channels are named SA , RA , SB , RB , which stands for send/receive Alice and send/receive Bob respectively. To abstract from the negotiation of a common generator g and a common group p , we assume that these are global parameters. These parameters are modeled with the variable G and are also known to a potential attacker.

```
role session(A, B: agent, Ka, AIK, Ks: public_key, H: function, G: text)
  def=
```

```
  local SA, RA, SB, RB: channel (dy)
```

```

composition
  challenger(A,B,Ka,Ks,H,G,SA,RA)
  /\ attestor (A,B,AIK,Ks,H,G,SB,RB)

end role

```

The environment role contains the global constants as well as a composition of sessions that the intruder is able to play as legitimate user.

```

role environment() def=

  const a, b : agent,
        aik, ka, ks, ki : public_key,
        h : function,
        g: text,
        na, nb, challenger_attestor_nb,
        attestor_challenger_na, sec_a_SK, sk1,nb1 : protocol_id

  intruder_knowledge = {a, b, aik, ka, ks, ki, inv(ki),
                       {i.ki}_(inv(ks)),g,h}

  composition
    session(a,b,ka,aik,ks,h,g)
  /\ session(a,i,ka,ki,ks,h,g)
  /\ session(i,b,ki,aik,ks,h,g)

end role

```

The following code fragment defines the security properties of our proposed protocol. The goal of the protocol is to authenticate Bob and to establish a secret key between both entities. The intruder should not be able to learn the secret key *SK*; furthermore he should not be able to perform a valid authentication.

```

goal
  authentication_on alice_bob_na
  authentication_on alice_bob_nb
  authentication_on Kab
  secrecy_of Kab

```

To specify the different security goals, we added in the respective state transitions the necessary events. These events are specified using the **witness**, **request** and **secret** structure (see [10], pp. 28 for more details). In our case, we use the **witness** and **request** for specifying the following security goals:

- *A* authenticates a genuine and authentic TPM on the value *Na*. This holds since only an authentic TPM is able to sign *Na* with a corresponding non-migratable key (*AIK* or special purpose key).

- A authenticates the TPM of B on the value K_{AB} . This holds since given the first statement, only the owner of the TPM possesses the corresponding private Diffie-Helman key. As a consequence, only A is able to compute the secret key based on the provided public key.
- A authenticates the TPM of B on the value Nb . This holds since given the second statement, only B can decrypt Nb and send it back to A .
- A and B share the key K_{AB} , which is confidential and is kept secret.

It should be noted that we do not directly authenticate B to A . A only determines whether she is currently communicating with the platform that has provided authentic measurements and that this channel is authentic.

For specifying authentication goals, AVISPA provides the *witness* and *request* commands and for specifying the secrecy goal, it provides the *secret* command. We again refer to the AVISPA technical documentation [10] for further information on these commands and security goals. Here, we will only exemplarily discuss how we specified authentication on Na .

For this purpose, the prover witnesses in state 1 the challenger that he answered his challenge.

```
/\ witness(B,A,challenger_attestor_na,Na')
```

This statement can be understood as follows: *Agent B declares that he wants Agent A agreeing on the value Na'*. The counterpart of this statement is required in state 4 and stated as:

```
/\ request(A,B,alice_bob_na,Na)
```

This statement is read as follows: *Agent A accepts the value Na and now relies on the guarantee that agent B exists and agrees with her on this value.*

Results

After specifying the protocol, we analyzed the model with the model checkers provided by AVISPA. We found no attack trace; thus the protocol analyzer reports that all security properties are satisfied and Protocol 3.5.2 is secure against Dolev-Yao attackers.

Protecting the Stored Measurement Log

To further enhance the security of our proposed protocol, we move transmitting the privacy-related SML into an cryptographically protected message. This has the advantage that the protocol must not necessarily be integrated into another cryptographic channel to satisfy Requirement 2. Our protocol then satisfies an additional security requirement (the secrecy of the SML). However, in that case, the challenger can only validate the platform configuration of B after receiving the last message. We also

modify messages (2), (3) and (4) of our protocol to provide complete confidentiality of all messages sent over the channel. This modification is comparable to the Finished messages as they are used in the TLS-protocol to detect potential tampering on the exchanged messages. The resulting protocol flow is shown in Protocol 3.6.3. We do not present the detailed protocol actions since they are rather similar to Protocol 3.5.2. The protocol was also modeled in AVISPA to formally analyze its security properties. Besides the security properties of Protocol 3.5.2, we analyzed whether the SML is always confidential and always protected by cryptographic measures. This security property is summarized through the following goal:

- A and B share the Stored Measurement Log (SML), which is privacy related and is confidential.

Protocol 3.6.3: Enhanced Robust Integrity Reporting Protocol

SUMMARY: B answers the attestation challenge of platform A

RESULT: Key establishment with key confirmation: confidential integrity reporting

1. *Notation.*

K_{AB} denotes the negotiated key between A and B .

2. *System setup.*

A must possess the certificate of the Privacy-CA to validate $Cert(AIK, K_{AIK})$.

3. *Protocol messages.*

$$A \rightarrow B : Na, g^a \bmod p, g, p \quad (1)$$

$$A \leftarrow B : Cert(AIK, K_{AIK}), \{Na, PCR, g^b \bmod p\}_{K_{AIK}^{-1}} \quad (2)$$

$$A \rightarrow B : \{Nb, g^a \bmod p\}_{K_{AB}} \quad (3)$$

$$A \leftarrow B : \{Nb, Na, SML, g^b \bmod p\}_{K_{AB}} \quad (4)$$

Results

After modeling the protocol and analyzing it with AVISPA, we found no attack trace; thus, the protocol analyzer reports that the enhanced version of the protocol satisfies all security properties and is secure against Dolev-Yao attackers. We also implemented both our proposed protocols (Protocol 3.5.2 and Protocol 3.6.3) and run performance

measurements on the computational overhead introduced through the additional cryptographic operations. The additional cryptographic operations increase the time for answering one attestation request by 0.70% compared to Protocol 3.3.2. We will present our detailed measurement results in Chapter 4 where we also discuss how we performed these measurements.

Protocol 3.6.3 does not only provide a means of establishing an authentic attestation channel, but also to establishes a cryptographic session key. Thus, it is not necessarily required to integrate the attestation protocol inside another cryptographic channel. We will discuss this property in the following section.

3.7 Attestation as TLS extension

We have shown that in order to prevent relay attacks against integrity reporting protocols, a second cryptographic channel needs to be established to ensure that the communication after the remote attestation is authentic, i.e., the challenger communicates with the system that provided the measurement values. In addition, according to Requirement 2, it is required that the attestation must not be performed over unprotected channels, since otherwise an adversary could eavesdrop on the communication channel and, thus, discover sensitive information about the platform configuration. Furthermore, for authenticating a particular entity it is necessary to use additional measures, e.g, TLS, that also establish an own cryptographic channel. However, one cryptographic attestation channel inside another cryptographic channel for authenticating a particular entity, clearly increases the complexity and one might argue whether this overhead is justifiable. We therefore propose to directly couple the attestation channel with the TLS-protocol. This approach enables extending the cryptographic channel to the TPM, thus, ensuring that the endpoint of the communication is the TPM. This combined process allows to authenticate a particular entity and to report the platform configuration in one step.

In the following, we will show how the TLS-protocol can be combined with our proposed robust integrity reporting protocol. We refer to the resulting protocol as Trusted-TLS, since it offers a secure way of performing integrity reporting without the need of establishing a second cryptographic channel. To ensure that the Trusted-TLS-protocol is applicable with mutual authentication as well as with one-way authentication, we propose two different protocols. Both proposed protocols were designed under the requirement that the existing TLS specification should be adapted as slightly as possible. The protocols cannot be directly combined with each other, since both entities must agree on the same protocol version. This is necessary since otherwise version rollback attacks would be basically possible.

3.7.1 Trusted-TLS with Anonymous User

The enhanced robust integrity reporting protocol uses similar mechanisms as the TLS-protocol. To combine both protocols, the *certificate verify* message of the TLS handshake protocol is created with the help of the TPM using the `TPM_Quote` command.

In that case, the *server hello* message serves as the *attestation challenge* message. To ensure that no man-in-the-middle is relaying a message, thus, trying to masquerade his platform configuration, the protocol finishes with the *Finished* messages. The resulting Protocol is shown in Protocol 3.7.2.

Protocol 3.7.2: TLS with anonym User

SUMMARY: *B* answers the attestation challenge of *A*

RESULT: Secure integrity reporting: authentication of *A*: key establishment with key confirmation

1. *Notation.*

Sid denotes the TLS security parameters and the protocol ID (`Trusted_DHE_anon`).

Px denotes the cryptographic engines which are supported by *X*

$Cert(X, K_X)$ is the public key certificate of *X*

DH_X denotes the public Diffie-Helman key of entity *X*

K_{AB} denotes the negotiated key between *A* and *B*

2. *System setup.*

A must acquire (and validate) the certificate of the Privacy-CA to validate

$Cert(AIK, K_{AIK})$. All entities must possess the certificate of the certification authority in order to authenticate the received certificates

3. *Protocol messages.*

- | | | | |
|-------------------|--|-----------------------|------|
| $A \leftarrow B$ | : B, Nb, Sid, Pb | (client hello) | (1) |
| $A \rightarrow B$ | : A, Na, Sid, Pa | (server hello) | (2) |
| $A \rightarrow B$ | : $Cert(A, K_A)$ | (server certificate) | (3) |
| $A \rightarrow B$ | : $DH_A, \{DH_A, Na, Nb, PCR\}_{K_A^{-1}}$ | (server key exchange) | (4) |
| $A \leftarrow B$ | : $Cert(AIK, K_{AIK})$ | (client certificate) | (5) |
| $A \leftarrow B$ | : DH_B | (client key exchange) | (6) |
| $A \leftarrow B$ | : $\{DH_A, DH_B, Na, Nm\}_{K_{AIK}^{-1}}$ | (certificate verify) | (7) |
| $A \rightarrow B$ | : $\{Finished\}_{K_{AB}}$ | (client finished) | (8) |
| $A \leftarrow B$ | : $\{Finished\}_{K_{AB}}$ | (server finished) | (9) |
| $A \leftarrow B$ | : $\{SML\}_{K_{AB}}$ | (attestation reply) | (10) |

4. *Protocol actions.*

- (a) *A* and *B* exchange randomly generated nonces Na, Nb and the supported cryptographic engines as well as protocol IDs including the protocol version. The protocol version should include the information that the `Trusted_DHE_anon` protocol is used.

- (b) A delivers the server certificate (3) to B .
 - (c) *DH-Key generation by A .* A selects an appropriate prime p and generator g of \mathbb{Z}_p^* ($2 \leq g \leq p-2$).
 - (d) A chooses a random secret a , $2 \leq a \leq p-2$, and computes $g^a \bmod p$. The public parts of the Diffie-Helman key are denoted as DH_A .
 - (e) A signs DH_A and transfers the message in step (4) to B .
 - (f) *DH-Key generation by B .* B chooses a random secret b , $2 \leq b \leq p-2$, and computes $g^b \bmod p$. The public parts of the Diffie-Helman key are denoted as DH_B .
 - (g) B signs all transferred messages with the *AIK* of the TPM and delivers the signature to A . For this purpose, the size of the transferred messages are shortened to match the 160-bit external data field of the `TPM_Quote` command.
 - (h) B and A compute the master secret based on the *attestation challenge* and the exchanged nonces and obtain K_{AB} . B and A then exchange the Finished messages in step (8) and (9).
 - (i) Finally, B transfers the Stores Measurement Log to A using the currently negotiated key.
-

3.7.3 Trusted-TLS with User Authentication

The TLS-protocol can also be used to mutually authenticate both entities. In that case, both entities need to possess a valid certificate and a corresponding private key that has been issued by a certification authority that is trusted for both entities. The TLS-specification dictates that the proof-of-knowledge of some secret authentication data is achieved by the *certificate verify* message. Since the *certificate verify* message is already used for the attestation reply message, a second *certificate verify* must be sent, which provides a proof-of-knowledge of a specific secret, i.e., the authentication data for authenticating a specific user. It is also possible to create a double signature with the *AIK* and another user specific key. This approach is implementation specific and provides no additional security, as we will see in the security analysis section. The resulting protocol for mutual authentication and integrity reporting is shown in Protocol 3.7.4. The modified messages compared to Protocol 3.7.2 are the messages (4'), (5'), (7') and (8'). They are marked with a ' in the following.

Protocol 3.7.4: TLS with user authentication

SUMMARY: B answers the attestation challenge of A

RESULT: Integrity reporting: mutual authentication: key establishment with key confirmation

1. *Notation.*

Sid denotes the TLS security parameters and the protocol ID (`Trusted_DHE_RSA`).

Px denotes the cryptographic engines which are supported by X

$Cert(X, K_X)$ is the public key certificate of X

DH_X denotes the public Diffie-Helman key of entity X

K_{AB} denotes the negotiated key between A and B

2. *System setup.*

A must acquire (and validate) the certificate of the Privacy-CA to validate

$Cert(AIK, K_{AIK})$. All entities must possess the certificate of the certification authority in order to authenticate the received certificates

3. *Protocol messages.*

$A \leftarrow B : B, Nb, Sid, Pb$	(client hello)	(1)
$A \rightarrow B : A, Na, Sid, Pa$	(server hello)	(2)
$A \rightarrow B : Cert(A, K_A)$	(server certificate)	(3)
$A \rightarrow B : DH_A, \{DH_A, Na, Nb\}_{K_A^{-1}}$	(server key exchange)	(4')
$A \leftarrow B : Cert(AIK, K_{AIK}), Cert(B, K_B)$	(client certificate)	(5')
$A \leftarrow B : DH_B$	(client key exchange)	(6)
$A \leftarrow B : \{DH_A, DH_B, Na, Nb, PCR\}_{K_{AIK}^{-1}}$	(certificate verify)	(7')
$A \leftarrow B : \{DH_A, DH_B, Na, Nb\}_{K_B^{-1}}$	(certificate verify)	(8')
$A \rightarrow B : \{Finished\}_{K_{AB}}$	(client finished)	(9)
$A \leftarrow B : \{Finished\}_{K_{AB}}$	(server finished)	(10)
$A \leftarrow B : \{SML\}_{K_{AB}}$	(attestation reply)	(11)

4. *Protocol actions.*

- (a) A and B exchange randomly generated nonces Na, Nb and the supported cryptographic engines as well as protocol IDs including the protocol version. The protocol version should include the information that the `Trusted_DHE_RSA` protocol is used.
- (b) A delivers the server certificate (3) to B .

- (c) *DH-Key generation by A.* A selects an appropriate prime p and generator g of \mathbb{Z}_p^* ($2 \leq g \leq p-2$).
 - (d) A chooses a random secret a , $2 \leq a \leq p-2$, and computes $g^a \bmod p$. The public parts of the Diffie-Helman key are denoted as DH_A .
 - (e) A signs DH_A and transfers the message in step (4') to B.
 - (f) B delivers its certificates in message (5') to A.
 - (g) *DH-Key generation by B.* B chooses a random secret b , $2 \leq b \leq p-2$, and computes $g^b \bmod p$. The public parts of the Diffie-Helman key are denoted as DH_B .
 - (h) B signs all transferred messages with the *AIK* of the TPM and delivers the signature to A (7'). For this purpose, the size of the transferred messages are shortened to match the 160-bit external data field of the `TPM_Quote` command.
 - (i) To additionally provide user-authentication B signs all transferred messages besides the PCR with a user specific key K_B^{-1} . This key is independent to the *AIK* and can be held in the protected storage of the TPM, e.g., as a Subject Key Attestation Evidence (SKAE) [169] or a key stored on a smart card. The signed message is transferred to A (8').
 - (j) B and A compute the master-secret based on the *attestation challenge* and the exchanged nonces and obtain K_{AB} . B and A then exchange the Finished messages in step (9) and (10).
 - (k) Finally, B transfers the Stores Measurement Log to A using the currently negotiated key (11).
-

3.8 Security Analysis

In this section, we will first discuss the security of our proposed TLS extension. We will then provide a formal security analysis of our proposed extension. For this purpose, we again use the AVISPA protocol prover [182].

3.8.1 Informal Security Analysis

The security of the TLS-protocol has extensively been analyzed in the literature (e.g., [117, 184, 64]). Thus, we will only informally discuss whether our proposed protocol is secure against the attack shown in Section 3.3.3.

If an attacker tries to masquerade his own platform configuration with the help of a second platform, it is required that the attacker relays the challenge represented by the *server hello* message to the second platform. This challenge is then signed together with the public DH-key of the second platform with the *AIK*. Since in that case, the public DH-key is part of the *certificate verify* message, the challenger will compute his

pre-master-secret based on this public DH-key. If the attacker, who is situated in the middle of the communication, modifies parts of the transferred public DH-parameter, both platforms would compute different Finished messages. This prevents the challenger from successfully validating the SML. The validation of the trust level of the platform configuration therefore fails.

Version rollback attacks on protocols of this type are possible as already discussed in Section 3.6.1. These attacks are characterized by the fact that an attacker tries to mask his public key together with the nonce provided by the verifier as new nonce for the prover. However, since the protocol's software integrity of the prover is also reflected in the platform configuration, the verifier will determine the platform configuration as not being trusted.

3.8.2 Formal Security Analysis

To formally analyze the proposed protocol, we again use the AVISPA protocol prover [182]. As already mentioned in Chapter 2, AVISPA is a model checker and not a theorem prover. Thus, we cannot provide a formal proof that the protocol is correct.

In the following we restrict us to only analyze Protocol 3.7.4. Protocol 3.7.2 can be modeled analogously. Each involved entity (i.e., prover and challenger), is modeled as a finite state machine and each transition from one state to another requires the receipt of a message and the sending of a reply message. Figure 3.3 shows the modeled finite state machine. For the sake of understanding, we divided in Figure 3.3 each state of client and server in two independent states S_i and S'_i with $i \in \{0, \dots, 5\}$. The transition to state S_i requires the receipt of a message and the transition from S'_i to S_{i+1} requires sending of a message. The transition from state S_i to S'_i is characterized by the computation overhead in order to compute and create the transmitted messages. In each state, the respective messages as specified in the protocols are transferred, e.g., the server receives in state S_1 message (1) of Protocol 3.7.4 and creates and delivers messages (2), (3) and (4) in state S'_1 to the client. The complete AVISPA source of Protocol 3.7.4 is presented in Section A.2 of the appendix.

We again use the Dolev-Yao intruder model [50] to model the attacker and the environment. To abstract from the negotiation of a common generator g and a common group \mathbb{Z}_p^* , we assume that these are global parameters known to all parties in the environment. However, this is not a security restriction since these messages can be transferred in plain-text without loss of security (see [45]).

We use AVISPA to verify the following security goals:

- A authenticates a genuine and authentic TPM on the value Na . This holds since only an authentic TPM is able to sign Na with a corresponding non-migratable key (AIK or special purpose key).
- A authenticates the TPM of B on the value K_{AB} . This holds since given the first statement, only the owner of the platform of B possesses the corresponding private Diffie-Helman key. As a consequence, only A is able to compute the secret key based on the provided public key.

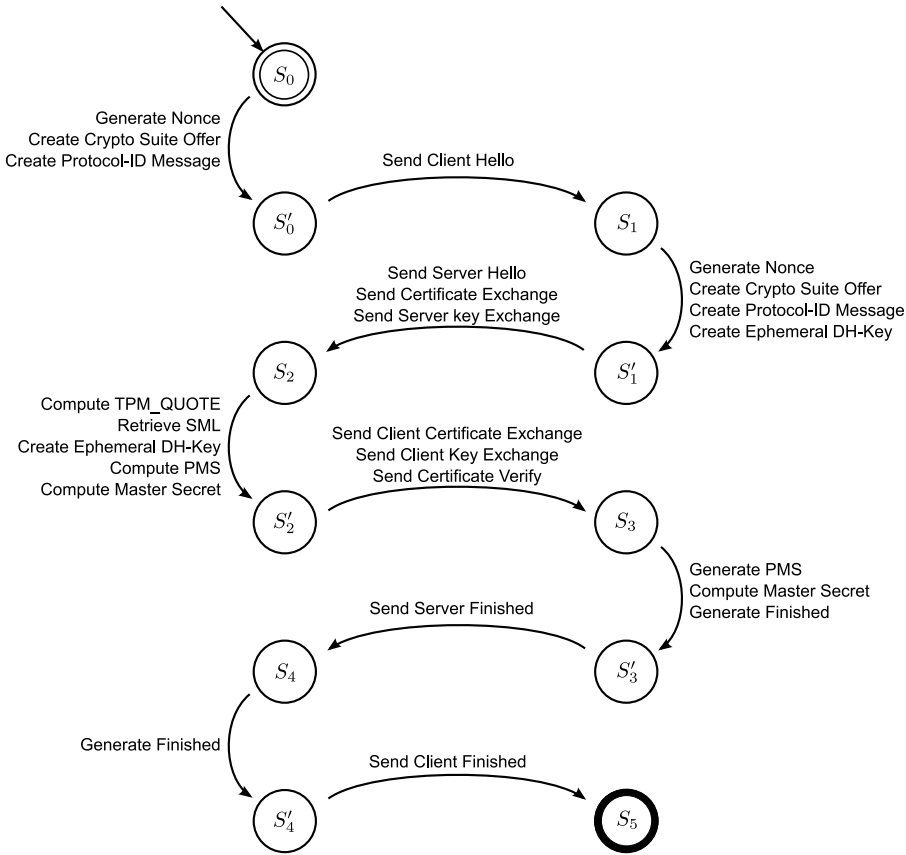


Figure 3.3: State machine of Client (platform B) and Server (platform A).

- A authenticates the TPM of B on the *Finished* messages. This holds since given the second statement, only B can encrypt the *Finished* messages and send it back to B.
- A authenticates B on the value Na . This holds since only B is able to sign Na using his own private key.
- B authenticates A on the value Nb . This holds since only A is able to sign Nb using his own private key.
- A and B share the key K_{AB} , which is confidential and is kept secret.
- A and B share the Stored Measurement Log (SML), which is privacy related and is kept secret.

Results After modeling the protocol, we analyzed the model with the different model checkers [16] provided by AVISPA. We found no attack trace; thus, the protocol analyzer

reports that all security properties are satisfied and Protocol 3.7.4 is secure against Dolev-Yao attackers.

3.9 Related Work

Since the specifications of the TCG are still in progress, many open issues exist. There is a large number of work focusing on the concepts of Trusted Computing. Sailer et al. [142] presented a comprehensive prototype based on Trusted Computing technologies. In particular, the authors come up with an architecture for integrity measurement, which contains an integrity reporting protocol that is consistent to the TCG-specification. We enhance this integrity reporting protocol in this chapter in order to make it resistant against masquerading attacks.

Balfe et al. [11] propose the integration of key exchange protocols into Direct Anonymous Attestation (DAA) [23] in P2P networks. However, the objectives of the integration of key exchange protocols are different to our proposal, since [11] aims at building stable identities in P2P networks. Additionally, the presented approach does not feature integrity reporting and cannot be directly applied to remote attestation.

Another related work is [70] which aims at building secure tunnels between endpoints. But this approach adds a new platform property certificate, which links the *AIK* to the TLS certificate. This platform property certificate can be a self-signed certificate and is then measured by the TPM and included in the SML. In contrast to that, our approach focuses on client attestation and does not need a modified way of how the measurements for the platform property certificate are made, since we directly bind the cryptographic channel to the *AIK*. In addition to that, the proposal by Goldman et al. is still vulnerable to the masquerading attack presented in this chapter.

Löhr et al. [103] present key components for scalable offline attestation. The approach is based on the TPM's ability to generate a non-migratable key which is bound to a specific platform configuration. The TPM bound key-pair is then certified and the certificate is integrated together with the SML inside an *attestation token*. The attestation token consists of all necessary information to make a statement whether a specific platform is trusted. It is public and can thus be shared amongst other entities. The solution proposed by Löhr et al. [103] is resistant against the masquerading attack presented in this chapter. However, our solution proposed in this chapter has a better performance compared since our solution does not require a new TPM bound key-pair to be generated each time the platform configuration changes.

Gasmi et al. [68] and Unger et al. [181] propose to include integrity information into secure channel establishment. For this purpose, they enhance the TLS-protocol with Subject Key Attestation Evidence (SKAE) certificates [169] that additionally carry integrity information. These approaches have similarities to our proposed TLS-extension. However, since they are based on the TPM's ability of binding a key to a specific platform configuration, they do not have a very good performance. That is caused by the fact, that each time the platform configuration changes, a new TPM key needs to be generated, which is bound to the current platform configuration. This approach has

therefore a smaller flexibility in particular if the platform configuration changes often over time. In addition, this approach would require a new mechanism which is able to reliably detect when the platform configuration changes and a new key needs to be generated. However, it is unclear how such a mechanism could be realized.

The TCG is also working on an own TLS-extension [176]. However, the specification for this extension has not been published yet and it can only be speculated how the exact security mechanisms are realized.

3.10 Summary

In this chapter, we analyzed the TCG-defined integrity reporting and extracted three requirements which must be satisfied in order to ensure secure attestation channels. We have shown in this chapter that the TCG-defined integrity reporting does not satisfy all requirements and is, thus, vulnerable to a masquerading attack. This does not only apply for the TCG-defined integrity reporting, but also to other approaches that provide integrity reporting and do not satisfy our extracted requirements. In the context of the TCG-defined integrity reporting, the problem evolved because the *AIK* does not reveal whether the attesting system is the one which really provides the measured values. This shortcoming is relevant in any scenario where remote attestation is used to guarantee the trust level of a system configuration. To address this issue, we have presented a robust protocol which prevents masquerading attacks. For that purpose, we added an authentication scheme into an integrity reporting protocol. The resulting protocol guarantees that the communication after the remote attestation is authentic, i.e., the challenger communicates with the system that provided the measurement values. Since this key agreement scheme increases the complexity, we also showed how the TLS-protocol can be extended in order to enable integrity reporting. This proposed protocol enables mutual authentication of two entities as well as attesting one entity to its platform configuration.

Chapter 4

Scalable Attestation

This chapter shows how the scalability of platform attestation can be improved. In this context, we propose three protocols that enable fast and secure integrity reporting for servers that have to handle many attestation requests. We implemented all of our protocols and compared them in terms of security and performance. Our proposed protocols enable a highly frequented entity to timely answer incoming attestation requests. Most of the material presented in this chapter has been previously published in: *Improving the Scalability of Platform Attestation* [160].

4.1 Introduction

The process of remote attestation and the requirements of protocols that support secure integrity reporting have been investigated in Chapter 3 and the literature (e.g., [152, 31, 164, 137, 103, 68]). In this context, all proposed solutions require the TPM to perform one expensive asymmetric cryptographic operation for each entity that is requesting integrity information. One reason for the high complexity is that, according to Requirement 1, a verifying entity has to ensure freshness of the integrity information, which is achieved by a challenge-response authentication involving the TPM. However, since the TPM is very limited in its computation power, the computation of one asymmetric operation takes between one and three seconds [138]. This causes the process of remote attestation to scale very poorly, which is particularly problematic if an entity is highly frequented and distributes integrity measurements to many clients (such as a server from which all clients request integrity information before they start using a particular service).

In this chapter, we present protocols that allow to overcome the performance bottleneck of a TPM, so that an entity is able to frequently report its integrity to many clients. To this end, we propose three different protocols that utilize different mechanisms of the TPM: the first protocol extends the most widely used platform attestation by bundling a number of attestation requests and answering them with one TPM operation; the second protocol requires a Trusted Third Party and utilizes hash-chains; and the third protocol realizes integrity reporting using time synchronized tick-stamps. Our

developed protocols do not require new Trusted Computing technology or modifications to the TPM specification. We have implemented all of our proposed protocols and have run simulations on currently available hardware TPMs. The simulations clearly indicate that the developed protocols are able to overcome the performance bottleneck of a TPM and can thus be used in environments where an entity is heavily exposed to integrity reporting queries.

The remainder of this chapter is organized as follows. In Section 4.2 we review related work, showing that all existing protocols scale poorly and thus cannot be used in highly frequented environments. Section 4.3 presents the assumptions and notations for our work. In Section 4.4 we present our protocols which enable a highly frequented entity to report its integrity. In Section 4.5 we analyze the security of our proposed solutions and in Section 7.7.2 we evaluate the protocols by looking at performance issues. Finally, we conclude in Section 4.7.

4.2 Evaluation of Existing Proposals

In this section, we analyze all known proposals and verify whether they can be used in highly frequented environments. We will show for each proposal that it scales poorly.

Löhr et al. [103] present key components for scalable offline attestation. In this context, they introduce an *attestation token* that consists of all necessary information to make a statement whether a specific platform is trusted. This attestation token is public and can thus be shared amongst other entities. However, to verify that a specific platform configuration is trusted and to obtain a proof that a specific attestation token belongs to a specific platform, expensive TPM operations are required. This concept is therefore not applicable in scenarios where frequent integrity verification is needed. As already briefly introduced in the introduction, this especially becomes problematic if a server wants to report its integrity to multiple clients.

Gasmi et al. [68] propose to include integrity information into secure channel establishment. For this purpose, they enhance the TLS-protocol with SKAE certificates [169] that additionally carry integrity information. However, their approach requires the computation of multiple expensive asymmetric cryptographic operations on the TPM on both endpoints. This again limits the usefulness in highly frequented environments due to massive performance degradations.

Shi et al. [152] address the time-of-use and time-of-attestation discrepancy of the TPM-based binary attestation. This discrepancy emerges because a software component is measured before execution and not directly before attestation. To overcome this problem, the authors propose to only attest to a small piece of sensitive code and thus measure a particular piece of code immediately before execution. However, the work requires that for every established communication channel, two `TPM_Seal`, one `TPM_Unseal` and one `TPM_Sign` operation are computed on the TPM, thus rendering the approach inapplicable in a client-server scenario.

Another approach to overcoming the bottleneck of a TPM is to use virtual TPMs [20]. A virtual TPM is a software TPM that only uses the underlying hardware TPM

for certain operations. This approach significantly increases the performance of the attestation process, since the CPU is used to calculate the attestation related operations. On the downside, because the TPM is implemented in software, this approach does not offer the same security as a hardware TPM. In particular, a software TPM does not provide functions such as active shields or active security sensors for preventing unauthorized access.

Another proposal [157] considers extending a TPM with hardware-based virtualization techniques and attaching the TPM to a faster bus or integrating it directly into the CPU. Since the TPM possesses much more computation power in this approach, performance degradation in the attestation process does not occur. However, the approach requires modifications of the TPM architecture.

Our work uses the TPM-based binary attestation which requires a trusted OS that performs measurements of all executed code, i.e., binary attestation. In contrast, [75] and [139] focus on semantic attestation based on attesting the behavior of software components. However, semantic attestation also requires a TCG-enhanced boot process to ensure that a small operating system kernel applies mechanisms that enforce the semantic attestation. To this end, a partially TPM-based binary attestation is required so that time degradations of the attestation process also apply to approaches based on semantic attestation.

Other related work, such as [31, 164, 137] additionally require that the communication perform expensive TPM operation during the execution of the attestation protocol.

4.3 Assumptions and Notations

In the following, we call the entity who wants to attest to the contents of its platform configuration the server and we call the entities that require an attestation the clients. The server is also referred to as prover, since he wants to prove to the verifier (a client) that he is in a trusted system configuration.

We assume that a trusted operating system, that is either based on a virtual machine monitor [67, 20, 156] or on a microkernel [68, 137, 101], performs integrity measurements on running software. In addition, this trusted operating system protects negotiated session keys against unauthorized extractions during normal runtime conditions. The architecture of one such trusted operating systems is also presented in Chapter 7 of this thesis. However, specifying and presenting the operating system environment is beyond the scope of this chapter.

We use the following notation: C_x denotes a client with $x \in \{1, \dots, n\}$ and S denotes a server; Na_x and Nb_x are random number nonces that are used to verify freshness and to detect impersonation. We denote a set of platform configuration registers as PCR and the Stored Measurement Log as SML .

4.4 Scalable Solutions

This section first highlights the shortcomings of the integrity reporting mechanisms as defined by the TCG [173]. We will then show how attestation can be used in highly frequented scenarios. In this context, we will present three different solutions that overcome the performance bottleneck of a TPM.

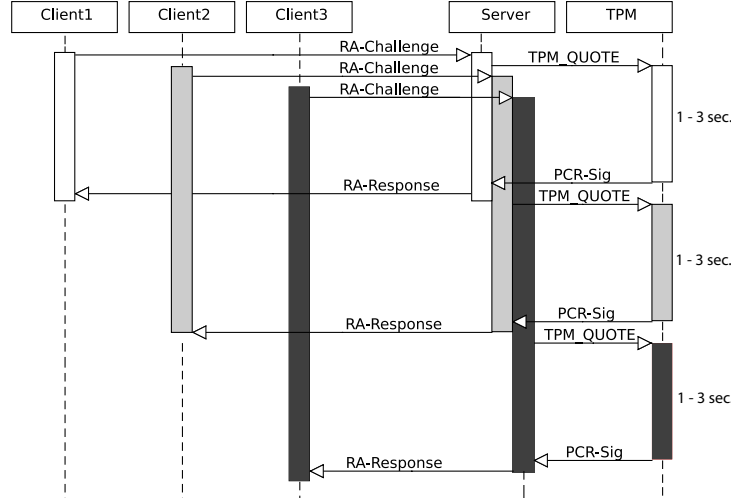


Figure 4.1: Delay of simultaneously arriving attestation requests

Figure 4.1 shows the delays that occur when multiple attestation requests arrive simultaneously at one server. Each attestation request includes one *nonce* that is generated by the challenger, to ensure freshness of the attestation reply. Every request can not be answered until the TPM has performed the `TPM_Quote` command, which signs a number of platform configuration registers and the nonce of the requester with an *Attestation Identity Key* (AIK). While the TPM performs internal computation, it is locked and does not accept any other request from the software stack. Thus, attestation requests are non-parallelizable. Incoming attestation requests are delayed until the TPM is able to process a new `TPM_Quote` command. We have implemented the integrity reporting protocol specified by the TCG and performed measurements on currently available TPMs (see Section 7.7.2). Our simulations on different hardware TPMs showed that a single `TPM_Quote` operation takes about one second, which verify the results presented in [138] stating that the time required for computing one `TPM_Sign` operation is between one and three seconds.

Figure 4.1 points out that even if a small number of clients require an attestation, massive delays will occur due to the sequential operation of the TPM. This fact can also result in denial of service attacks since a client may not receive a valid attestation response from the server in the expected period of time. As a result, this shortcoming could be exploited by an adversary to prevent waiting clients from successfully communicating with the server.

That is not only the case when the `TPM_Quote` command is used, but also when integrity reporting is realized implicitly through other TPM-operations, e.g., through *sealing* [96] or *binding* [103, 137]. We also ran simulations of an attestation protocol that utilizes `TPM_CertifyKey` combined with `TPM_Unbind` as proposed by Löhr et al. [103] and Sadeghi et al. [137]. These protocols enable platform attestation by sealing a non-migratable TPM key to a set of platform configuration registers. However, the time for computing one `TPM_Unbind` operation (around one second¹) is comparable to computing one `TPM_Quote` operation. These results clearly indicate that computing asymmetric cryptography on a TPM is very expensive, making currently proposed attestation protocols hardly applicable in highly frequented scenarios.

4.4.1 Multiple-Hash Attestation

The Multiple-Hash Attestation protocol extends the integrity reporting defined by the TCG by bundling a number of attestation requests and answering them with one `TPM_Quote` operation. For this purpose, we utilize the properties of a collision resistant hash function. In particular, we map a number of incoming attestation requests to a single request whose nonce is computed as a hash of all nonces of the arriving attestation requests. Before the nonces are passed to the TPM, they are also added to a *NonceList* that describes which nonces were hashed to produce the target nonce. The prover then transmits the output of the `TPM_Quote` command together with the *NonceList* and the *SML* to all entities that issued an attestation request. It should be noted that the TPM simply signs all PCRs and not only a set of the PCRs. Since the TPM simply concatenates all PCR values and then creates a signature on all values, the receiving clients can also easily verify the values of a subsequent set of the PCRs (Compare Chapter 5.1 for additional information on this issue). Before accepting the proof, the verifier checks whether his nonce is part of the *NonceList* transmitted alongside the `TPM_Quote` output. This approach satisfies Requirement 1 of a secure attestation channel and provides a proof for the freshness of integrity information.

To prevent masquerading attacks on the authenticity of the platform configuration [164] and to satisfy Requirement 3 of a secure attestation channel, it is necessary to integrate an authentication process in the attestation protocol. This authentication process is realized (similar to Section 3.6) by adding a key-establishment in order to ensure that the channel of attestation is authentic. The negotiated key can then be used as an encryption key for all subsequent messages sent between server and client. This mechanism also guarantees an end-to-end communication and keeps the attestation channel from becoming compromised by another application that could take over the attestation channel after the attestation has succeeded. Requirement 2 of a secure attestation channel is satisfied by delivering the *SML* in the established cryptographic channel. Protocol 4.4.2 shows the resulting protocol flow.

¹All measurements were performed on a TPM ST Microelectronics ST19WP18-TPM-C and an Atmel TPM 1.2

Protocol 4.4.2: Multiple-Hash Attestation

SUMMARY: S answers the attestation challenge of C_x

RESULT: Integrity reporting: key establishment with key confirmation

1. *System setup.*

C_x must acquire (and validate) the certificate of the Privacy-CA to validate $Cert(AIK, K_{AIK})$

2. *Protocol messages.*

$$C_x \rightarrow S : Na_x, g^{c_x} \bmod p \quad (1)$$

$$C_x \leftarrow S : Cert(AIK, K_{AIK}), NonceList, \\ g^s \bmod p, \{HashedNonceList, PCR\}_{K_{AIK}^{-1}} \quad (2)$$

$$C_x \rightarrow S : \{Nb_x, g^{c_x} \bmod p\}_{K_{SC_x}} \quad (3)$$

$$C_x \leftarrow S : \{Na_x, Nb_x, SML, g^s \bmod p\}_{K_{SC_x}} \quad (4)$$

3. *Protocol actions.*

- (a) *Precomputation by S .* S selects an appropriate prime p and generator g of \mathbb{Z}_p^* ($2 \leq g \leq p-2$). S chooses a random secret s , $2 \leq s \leq p-2$, and computes $g^s \bmod p$. S transmits p and g to all C_x .
- (b) *Precomputation by C_x .* C_x chooses a random secret c_x , $2 \leq c_x \leq p-2$, and computes $g^{c_x} \bmod p$.
- (c) *Attestation challenge.* $C_x, x = 1, \dots, n$, choose a non-predictable nonce Na_x and transmit message (1) to S .
- (d) S adds Na_1, \dots, Na_n to the *NonceList* and computes:

$$NonceList = Na_1 || Na_2 || \dots || Na_n \quad (4.1)$$

$$HashedNonceList = h(h(Na_1, g^s \bmod p) || \\ h(Na_2, g^s \bmod p) || \dots || h(Na_n, g^s \bmod p)) \quad (4.2)$$

- (e) S then computes the attestation response message by signing the *Hashed-NonceList* and the *PCRs* using an *AIK*. S then transmits message (2) to C_x .
- (f) *Key confirmation.* C_x computes the shared session key by computing $K_{SC_x} = (g^s)^{c_x} \bmod p$. C_x then generates a second non-predictable nonce (Nb_x) and transfers message (3) to S :

- (g) S computes the shared session key by computing $K_{SC_x} = (g^{c_x})^s \bmod p$ and decrypts the received message with K_{SC_x} . S then transfers message (4) to C_x .
- (h) C_x verifies the signature of $\{HashedNonceList, PCR\}_{K_{AIK}^{-1}}$ and checks the freshness of Na_x , by recalculating $HashedNonceList$ using the transmitted $NonceList$. Based on the received SML and the PCR values C_x processes the SML and re-computes the received PCR values. If the computed values match the signed PCR values, the SML is valid and untampered. Finally, C_x has to verify that the delivered integrity reporting values match the given reference values; thus C_x can decide if S is in a trusted system state.

The above solution works well if several attestation requests arrive at once; if they are slightly delayed, the Multiple-Hash Attestation Protocol can be optimized further. For this purpose, we introduce a ring buffer with three different areas, each consisting of one area for the *NonceList* and another area for the output of a TPM operation. Figure 4.2 depicts this tri-state ring buffer and shows how it is embedded in the attestation protocol. This tri-state ring buffer holds all relevant data for three different threads that are responsible for passing data from the verifier to the TPM and vice versa:

- Input thread: This thread collects challenge-requests and writes them to a *NonceList* area of the ring buffer.
- Working thread: This thread computes the *HashedNonceList* from the *NonceList* stored in one of the three areas of the buffer and executes the `TPM_Quote` on the *HashedNonceList* and the *PCR*. The result is then written back to the same area of the buffer and the ring buffer is rotated.
- Output thread: This thread is responsible for reading the results of the TPM operation from the buffer and for delivering the results to the corresponding clients who requested attestation.

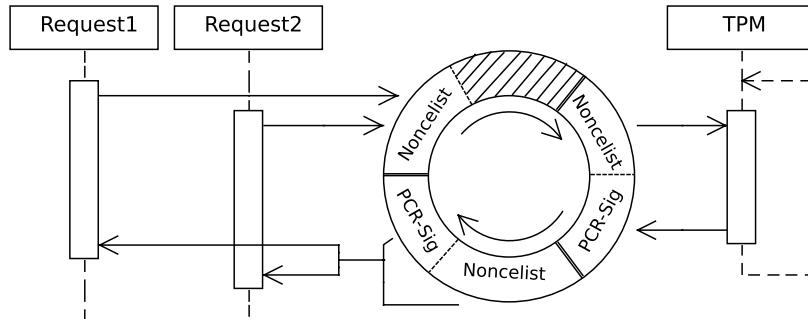


Figure 4.2: Tri-state ring buffer of the Multiple-Hash Attestation

Each time a new attestation request arrives at the server, the input thread forks a new process. This process adds the incoming nonce to the *NonceList* and remembers the area of the ring buffer that is responsible for the process. The working thread runs continuously; this thread is responsible for passing data to and receiving data from the TPM. As soon as the working thread receives an answer from the TPM, the working thread writes the results in the ring buffer and rotates the tri-state ring buffer. This working thread then retrieves the new *NonceList* of the adjacent area, which he hashes to create the *HashedNonceList* and passes to the TPM. While the working thread is operating on the TPM, the processes that are responsible for answering the client's attestation-requests are waiting until the tri-state buffer has been rotated one round and their data is ready. As soon as the requested data is present in the buffer, the processes that are responsible for the attestation-request can deliver the attestation-response, consisting of the signed *HashedNonceList* and *PCR*, to the requester.

This approach enables a very efficient realization of the Multiple-Hash Attestation. Since the working thread that is responsible for the TPM is also responsible for the ring buffer, a maximum utilization of the TPM can be achieved. Incoming attestation requests must wait a maximum of two rotation steps of the ring buffer before the result is available. Thus, in the worst case, the response is available after two signature computations on the TPM have been performed.

4.4.3 Timestamped Hash-Chain Attestation

Another alternative to reduce the number of costly TPM operations, is to relinquish the server from integrating every nonce of each client into the costly TPM operations. This can be achieved by involving a Trusted Third Party (TTP) that is responsible for issuing nonces for the attestation process. For this purpose, the TTP provides nonces, signatures of these nonces and timestamps that state when the TTP generated the nonces. We divide the time into intervals; each nonce together with the timestamp is associated to a particular time interval. To ensure an attestation request arrives at the server in a particular interval, the server uses the nonce which is valid in the current interval and not a nonce provided by the client. The TTP and all clients need to be loosely time-synchronized to enable the clients to verify the validity of the TTP generated nonce for a certain interval.

A straightforward implementation of this idea results in a relatively high load at the TTP. During i intervals, the TTP has to generate $n * i$ nonces and timestamps to serve n servers. Since the intervals should be in the range of seconds, a TTP has to generate new nonces very frequently. To relieve the TTP from generating a nonce for each interval, we propose using hash-chains [98]. The TTP issues a nonce Na_0 with time-stamp only for the first time interval and the server uses the initial TTP-generated nonce Na_0 only for the first attestation query. After each interval the server performs a hash-operation on the nonce by applying the hash function h successively to the nonce of the previous interval in order to produce the nonce of the next interval, i.e., $Na_{v+1} = h(Na_v)$, with $v = 0, 1, \dots, k$.

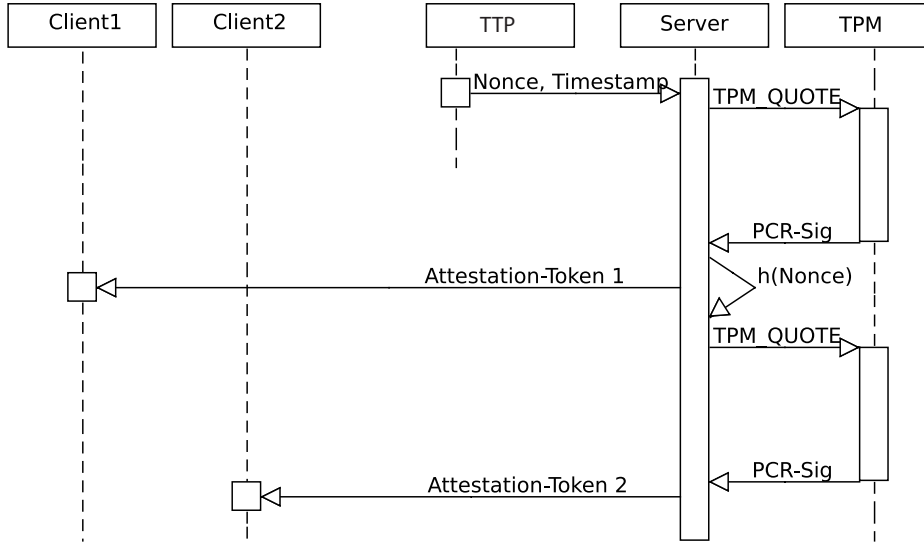


Figure 4.3: Protocol flow of the Timestamped Hash-Chain Attestation

Since the TTP issues a timestamp only for the first nonce, the verifying clients cannot directly validate whether the received nonce is valid in the current interval. To enable the clients to verify the validity of the nonce, the server has to provide a proof that he applied the hash function the correct number of times. This ensures freshness of integrity information and thus satisfies Requirement 1 of a secure attestation channel. Since the attestation service of the server is part of the server's platform configuration and thus its state is included in the *SML*, the clients can verify that the attestation service is in a trusted state. The proof is thus being made through validating the platform configuration of the service.

Figure 4.3 depicts the Timestamped Hash-Chain Attestation. After each interval, the server calculates a new hash-value with which the new attestation is performed. The length of such an interval depends on how long it takes to create an attestation token (which is TPM related). As a result, one interval cannot be smaller than it takes to generate an attestation token. In Section 7.7.2, we present some evaluation results of a representative TPM. On our used TPM hardware, one interval cannot be smaller than approx. 1.4 seconds.

To enable the clients to validate the freshness of the platform configuration, the server delivers an attestation token (τ) to the requester. This attestation token consists of the TTP-signed seed nonce together with a timestamp, an *AIK* signed message with the current interval count, the nonce of the current interval, the *PCRs* and the public Diffie-Helman key of the server and the certificate of the *AIK*. We denote the initial TPM-signed nonce as seed nonce since this initial nonce is generated by the TTP and all further nonces are computed based on this nonce. The attestation token therefore

consists of all information that is necessary to validate the freshness of the attestation response:

$$\begin{aligned} \tau = & \{Na_0, time_0\}_{K_{TTP}^{-1}}, Cert(AIK, K_{AIK}) \\ & \{h^v(Na_0), v, PCR, g^s \bmod p\}_{K_{AIK}^{-1}}. \end{aligned} \quad (4.3)$$

The attestation token introduced here has similarities to the one introduced in [103]. However, the main difference is that a verifier can validate the platform configuration without requiring the prover to perform expensive TPM operations. To verify that the platform configuration of the server is trusted, the clients have to verify the validity of all certificates, all signatures and the validity of the timestamp. Note that this cost intensive validations must only be performed by the clients and not by the server. Furthermore, the clients have to verify whether the received nonce is valid in the current active interval. For this purpose, the following equation must hold: $time_0 + v \cdot t \leq now \leq time_0 + (v + 1) \cdot t + \epsilon$, where v represents the interval number, t the time length of the interval, and $time$ the point of time when the TTP generated the *nonce*. Moreover, *now* denotes the current time of the verifier and ϵ a certain error range. The time length of the interval is part of the *SML* and thus represented by the platform configuration of the server. The protocol again integrates an authentication process to ensure Requirement 3 of a secure attestation channel. Requirement 2 of a secure attestation channel is again satisfied by delivering the *SML* in the established cryptographic channel. The resulting protocol is shown in Protocol 4.4.4.

Protocol 4.4.4: Timestamped Hash-Chain Attestation

SUMMARY: S answers the attestation challenge of C_x

RESULT: Integrity reporting: key establishment with key confirmation

1. *System setup.*

C_x must acquire (and validate) the certificate of the Privacy-CA to validate $Cert(AIK, K_{AIK})$

2. *Protocol messages.*

$$TTP \rightarrow S : \{Na_0, time_0\}_{K_{TTP}^{-1}} \quad (1)$$

$$C_x \rightarrow S : g^{c_x} \bmod p \quad (2)$$

$$C_x \leftarrow S : \{Na_0, time_0\}_{K_{TTP}^{-1}}, \{h^v(Na_0), v, g^s \bmod p, PCR\}_{K_{AIK}^{-1}}, \quad (3)$$

$$Cert(AIK, K_{AIK})$$

$$C_x \rightarrow S : \{Nb_x, g^{c_x} \bmod p\}_{K_{SC_x}} \quad (4)$$

$$C_x \leftarrow S : \{Na_v, Nb_x, SML, g^s \bmod p\}_{K_{SC_x}} \quad (5)$$

3. Protocol actions.

- (a) S chooses a TTP for providing the seed nonce and the TTP transfers message (1) to S .
- (b) *Precomputation by S .* S selects an appropriate prime p and generator g of \mathbb{Z}_p^* ($2 \leq g \leq p - 2$). S chooses a random secret s , $2 \leq s \leq p - 2$, and computes $g^s \bmod p$. S transmits p and g to all C_x .
- (c) *Precomputation by C_x .* C_x chooses a random secret c_x , $2 \leq c_x \leq p - 2$, and computes $g^{c_x} \bmod p$.
- (d) *Attestation challenge.* C_x , $x = 1, \dots, n$, deliver message (2) to S .
- (e) Depending on the current interval v , S computes the current valid nonce Na_v by calculating:

$$Na_v = h(Na_{v-1}), \quad \text{with } v = 1, \dots, k. \quad (4.1)$$

- (f) S computes the attestation response message by signing Na_v , the current interval v and the set of requested $PCRs$ using an AIK thereby obtaining $\{h^v(Na_0), v, g^s \bmod p, PCR\}_{K_{AIK}^{-1}}$.
- (g) S transmits the attestation token τ in message (3) to C_x .
- (h) *Key confirmation.* C_x computes the shared session key by computing $K_{SC_x} = (g^s)^{c_x} \bmod p$. C_x then generates a second non-predictable nonce (Nb_x) and transfers message (4) to S .
- (i) S computes the shared session key by computing $K_{SC_x} = (g^{c_x})^s \bmod p$ and decrypts the received message with K_{SC_x} . S then transfers message (5) to C_x .
- (j) C_x verifies all signatures and checks the freshness of the Na_v by checking whether Equations (4.2) and (4.3) hold:

$$time_0 + v \cdot t \leq now \leq time_0 + (v + 1) \cdot t + \epsilon, \quad (4.2)$$

$$h^v(Na_0) \stackrel{?}{=} Na_v. \quad (4.3)$$

- (k) Finally, C_x verifies that the platform configuration of S is trusted based on the SML and the $PCRs$.

4.4.5 Tickstamp Attestation

The tickstamp attestation uses the tick-counter of a TPM (a tick is in the time range of milliseconds up to seconds). A TPM provides a mechanism to create a signature of the current tick-counter value. The resulting data structure includes a signature of the

current tick-counter value and the time interval after which the counter is periodically incremented.

To use tick-counters during platform attestation, we utilize the concept that a TPM enables the creation of non-migratable keys that are bound to the platform configuration. Before a TPM performs platform attestation, a non-migratable key is generated on the TPM and bound to a specific set of platform configuration registers. This key is certified using an *AIK*, which gives a proof that the key is bound to a specific set of platform configuration registers. The bound key is then used in periodic time intervals to generate *TickStampBlobs*, which is only possible while the platform configuration has not changed. A *TickStampBlob* consists of a complete `TPM_CURRENT_TICKS` structure [168] and the resulting signature. It includes the current value of the tick-counter, a tick-session identifier, and a signature of the data. To demonstrate to a remote party that the platform configuration is to be trusted, an *attestation token* τ is computed. This token comprises the latest *TickStampBlob*, which is the first part of the token shown in Equation (4.4), as well as the certificate of the key used to generate the *TickStampBlob*:

$$\begin{aligned} \tau = & \{ \text{currentTicks}, g^s \bmod p \}_{K_{TS}^{-1}}, \\ & \text{Cert}(TS, K_{TS}), \text{Cert}(AIK, K_{AIK}). \end{aligned} \quad (4.4)$$

The trust level of the platform configuration can be evaluated based on the certificate and the tick-count value inside the *TickStampBlob*. However, the token gives only an assertion about the platform configuration in relation to the tick-counter on the platform that wants to demonstrate its configuration. A synchronization of the tick-counter of the challenger and the prover is thus needed.

In the following, we will discuss two means of achieving this synchronization. Most implementations of the TPM specification initialize a tick-session after a reboot of the system with the value zero. To uniquely identify a specific tick-session, the TPM also adds a nonce to the tick-session. Since the nonce is also part of every *TickStampBlob*, two different *TickStampBlob* structures can be uniquely related to their session.

Challenge-Response Synchronization The easiest way to perform synchronization with the tick-session of the server is to deliver a nonce to the server which is then signed with `TPM_TickStampBlob`. The resulting signature includes the complete `TPM_CURRENT_TICKS` structure [168], which gives an assertion about the actual tick-counter value, the tick-rate and the identifying nonce of the current tick-session. Based on this information, the verifier can check the freshness of the actual attestation token. However, this concept requires one expensive sign operation on the TPM, which does not scale. It is thus not applicable in highly frequented environments.

Time Synchronization using a TTP Another alternative is to involve a Trusted Third Party in the synchronization protocol. This sync-TTP is responsible for associating a specific tick-session to a specific global time. For this purpose, the TTP generates

and transfers a nonce to the server, directly after the tick-session on the server is initialized. This nonce is then signed with `TPM_TickStampBlob` by the server TPM and the result is delivered to the sync-TTP. The sync-TTP verifies the signatures and generates a synchronization token that includes the global time, the round-trip time and the received result. This sync token is then returned to the server which adds the token to all generated attestation tokens.

Based on the synchronization token, an association between the global time and the beginning of a specific tick-session is made. A client only needs to synchronize his time with the sync-TTP in order to make a statement about the freshness of the received attestation token τ in Equation (4.4) and to satisfy Requirement 1 of a secure attestation channel. The sync-TTP thus provides services similar to a generic NTP server. It is also reasonable that a NTP server can be extended with the ability to create synchronization tokens, since these protocol steps only require minimal additional computations. Tickstamp Attestation with TTP-based time synchronization is shown in Protocol 4.4.6. As the preceding protocol, Protocol 4.4.6 integrates an authentication process to ensure Requirement 3 of a secure attestation channel. Requirement 2 is again satisfied by delivering the SML in the established cryptographic channel.

Protocol 4.4.6: Tickstamp Attestation with synchronization token

SUMMARY: S answers the attestation challenge of C_x

RESULT: Integrity reporting: key establishment with key confirmation

1. *System setup.*

C_x must acquire (and validate) the certificate of the Privacy-CA to validate $Cert(AIK, K_{AIK})$

2. *Protocol messages of the initialization phase.*

$$TTP \rightarrow S : Nt \quad (I)$$

$$TTP \leftarrow S : \{Nt, currentTicks\}_{K_{TS}^{-1}}, Cert(TS, K_{TS}), \quad (II)$$

$$Cert(AIK, K_{AIK})$$

$$TTP \rightarrow S : \{\{Nt, currentTicks\}_{K_{TS}^{-1}}, Time\}_{K_{TTP}^{-1}} \quad (III)$$

3. *Protocol messages of the attestation phase.*

$$C_x \rightarrow S : g^{c_x} \bmod p \quad (1)$$

$$C_x \leftarrow S : Cert(TS, K_{TS}), Cert(AIK, K_{AIK}), \tau_{sync}, \quad (2)$$

$$\{currentTicks, g^s \bmod p\}_{K_{TS}^{-1}}$$

$$C_x \rightarrow S : \{Nb_x, g^{c_x} \bmod p\}_{K_{SC_x}} \quad (3)$$

$$C_x \leftarrow S : \{Nb_x, SML, g^s \bmod p\}_{K_{SC_x}} \quad (4)$$

4. *Protocol actions of the initialization phase.*

- (a) S selects a TTP for providing a sync-token. The TTP transfers a nonce Nt using message (I) to S .
- (b) S creates a non-migratable TPM key (K_{TS}) that is bound to a specific set of platform configuration registers. S certifies K_{TS} with K_{AIK}^{-1} . The resulting structure is denoted as $Cert(TS, K_{TS})$ and gives an assertion to which $PCRs$ K_{TS} is bound. S then signs the actual tick-counter value with K_{TS}^{-1} and delivers message (II) to the TTP.
- (c) The TTP verifies the signature and creates a time stamp on the received message and transfers the sync-token (τ_{sync}) in message (III) to S .

The server has now received a sync-token that can subsequently be used by the clients to verify freshness of the attestation token. After completing this initialization phase, the server is ready to answer attestation requests.

5. *Protocol actions of the attestation phase.*

- (a) *Precomputation and Pre-deployment by S .* S selects an appropriate prime p and generator g of \mathbb{Z}_p^* ($2 \leq g \leq p-2$). S chooses a random secret s , $2 \leq s \leq p-2$, and computes $g^s \bmod p$. S transmits p and g to all C_x .
 - (b) *Precomputation by C_x .* C_x chooses a random secret c_x , $2 \leq c_x \leq p-2$, and computes $g^{c_x} \bmod p$.
 - (c) *Attestation challenge.* $C_x, x = 1, \dots, n$, deliver message (1) to S .
 - (d) The server periodically signs the actual tick-counter value with K_{TS}^{-1} . The resulting data structure is denoted as *TickStampBlob*.
 - (e) S transmits the synchronization token $\tau_{sync} = \{\{Nt, currentTicks\}_{K_{TS}^{-1}}, Time\}_{K_{TTP}^{-1}}$ and the attestation token τ in message (2) to C_x .
 - (f) *Key confirmation.* C_x computes the shared session key by computing $K_{SC_x} = (g^s)^{c_x} \bmod p$. C_x then generates a second non-predictable nonce (Nb_x) and transfers message (3) to S .
 - (g) S computes the shared session key by computing $K_{SC_x} = (g^{c_x})^s \bmod p$ and decrypts the received message with K_{SC_x} . S then transfers message (4) to C_x .
 - (h) Finally, C_x verifies all signatures and checks whether the attestation token τ is fresh using the synchronization token τ_{sync} . In addition, C_x verifies that the platform configuration of S is trusted based on the *SML* and the *PCRs*.
-

4.5 Security Analysis

In this section, we will discuss the security of our proposed protocols. Since all protocols include an authentication step, we will first analyze whether this step is secure and enables secure integrity reporting. We finally perform a formal verification using the AVISPA protocol prover [182].

4.5.1 Security of the Authentication

This section first discusses the security of the authentication step by looking at potential attacks, including man-in-the-middle and version rollback attacks.

Man-in-the-Middle attack

All presented protocols prevent an attacker from hiding his malicious software configuration by performing a relay attack [164, 70], since all subsequent messages are encrypted with the computed session key K_{SC_x} . It is not possible for an attacker to perform some sort of man-in-the-middle attack and to establish two different cryptographic sessions between the verifier and the challenger, as he is not able to modify the attestation response of the prover. Since the session key K_{SC_x} is protected by the trusted operating system, (e.g., by storing it in a special purpose region of a security kernel or in a special virtual machine) it is not possible to extract this key under normal run-time conditions. Changing the trusted operating system environment to enable extraction of this key would lead to a non-conformant system state which would have been detected in the attestation phase.

Version rollback attack

In this attack scenario, three different parties, a verifier, a prover and an adversary, are involved. The adversary tries to relay the attestation challenge of the verifier to the prover, thus trying to masquerade a trusted system configuration. The verifier and the adversary run an authentication enhanced attestation protocol (e.g., Protocol 4.4.2), while the prover runs the TCG-defined attestation protocol. This attack can be classified as a version rollback attack, in which the adversary masks his public key together with the nonce provided by the verifier as new nonce for the prover. The prover does not verify the syntax of the attestation challenge. He directly signs the masqueraded nonce including public key and delivers the computed signature to the adversary. The adversary simply forwards the obtained message and both adversary and verifier compute the shared key based on the exchanged public keys. However, since the protocol's software integrity of the prover is also reflected in the platform configuration, the verifier will determine the platform configuration as not being trusted. The version rollback attack is therefore only a theoretical attack and fails, since it is not possible to successfully masquerade a trusted system configuration.

4.5.2 Formal Security Analysis

In the following we restrict us to analysis of Protocol 4.4.2. All other protocols can be handled analogously. Each involved entity (i.e., prover and verifier), is modelled as a finite state machine and each transition from one state to another requires the receipt of a message and the sending of a reply message. Thus, the verifier is modelled as a state machine with three states and the prover is modelled as a state machine with two states. In each state, the respective message as specified in the protocols are transferred, e.g., the prover receives in its first state message (4) of Protocol 4.4.2 and creates and delivers message (5) to the verifier.

We use the Dolev-Yao intruder model [50] to model the attacker and the environment. In this intruder model the attacker has full control over all messages that are sent over the network. The attacker can therefore intercept, analyze or modify messages, as well as compose new messages and send the messages to any party. To abstract from the negotiation of a common generator g and a common group \mathbb{Z}_p^* , we assume that these are global parameters known to all parties in the environment. However, this is not a security restriction since these messages can be transferred in plain-text without loss of security (see [45]).

We use AVISPA to verify the following security goals:

- C_x authenticates a genuine and authentic TPM on the value Na_x . This holds since only an authentic TPM is able to sign Na_x with a corresponding non-migratable key (AIK or special purpose key).
- C_x authenticates the TPM of S on the value K_{SC_x} . This holds since given the first statement, only the owner of the TPM possesses the corresponding private Diffie-Helman key. As a consequence, only C_x is able to compute the secret key based on the provided public key.
- C_x authenticates the TPM of S on the value Nb_x . This holds since given the second statement, only S can decrypt Nb_x and send it back to C_x .
- C_x and S share the key K_{SC_x} , which is confidential and is kept secret.
- C_x and S share the Stored Measurement Log (SML), which is privacy related and is kept secret.

It should be noted that we do not directly authenticate S to C_x . C_x only determines whether he is currently communicating with the platform that has provided authentic measurements and that this channel is authentic.

After modelling the protocol, we analyzed the model with the model checker provided by AVISPA. We found no attack trace; thus the protocol analyzer reports that all security properties are satisfied and Protocol 4.4.2 is secure.

4.5.3 Security Considerations of the Multiple-Hash Attestation

The main difference between the existing TCG-defined integrity reporting and the Multiple-Hash Attestation is that multiple nonces are hashed to one single nonce. The security of this process relies on the property that the hash function, in our case SHA-1, is collision resistant. If a collision resistant hash function is used, it is infeasible for an adversary to find a collision that can be used to masquerade a trusted system configuration.

4.5.4 Security Considerations of the Timestamped Hash-Chain Attestation

A general problem in the process of platform attestation is that an unrefresh platform configuration can be replayed, leading to a non-trusted platform configuration being masqueraded as trusted. To provide protection against this attack, Requirement 1 of a secure attestation channel was formulated which must be satisfied. This is realized by using randomly generated nonces combined with the challenge-response authentication method. However, in the context of the Timestamped Hash-Chain Attestation, these nonces are derived from one seed nonce by applying a hash function on this value. It is therefore possible to generate nonces that are valid in the future. This property can be exploited by an adversary by generating nonces that are valid in future intervals, computing the attestation token in a trusted configuration, and replaying the computed attestation token in the future after compromising the platform.

The risk that an adversary may perform such an attack can be minimized by preventing an adversary's ability to inject nonces corresponding to future time intervals. That can be done, for example, by modifying the operating system so that it only allows certain trusted processes to communicate with a TPM. These trusted processes should only accept seed nonces with a valid signature that have been created by a TTP. Since the configuration of the operating system is also part of the platform configuration, a verifier can check whether the prover's OS is in a trusted state and thus possesses a mechanism to prevent attacks of this type.

4.5.5 Security Considerations of the Tickstamp Attestation

The security of the Tickstamp Attestation relies on the assurance that a specific non-migratable TPM key, satisfying certain criteria, is used. This assurance is made using a certificate generated through `TPM_CertifyKey`. Löhr et al. [103] verified that the concept of binding a key to a specific set of platform configuration registers is secure against man-in-the-middle attacks. We will thus only look at the differences between the protocol proposed in [103] and our proposal. In contrast to Löhr et al., we also integrate a public Diffie-Helman key into the K_{TS} signed message. Using K_{TS} as a signing key at a specific time is only possible if the platform configuration is in a known and trusted state. The extraction of the Diffie-Helman key requires a modified system configuration that causes the state of the platform to change. The TPM will then deny decryption of the sealed key K_{TS} . To further enhance security, the Diffie-Helman key should also be

held in a special purpose region of a microkernel or virtual machine, as, for example, proposed in [68]. It should also be noted that each time K_{TS} is used, it is verified that the actual platform configuration is consistent with the platform configuration K_{TS} was bound to.

4.6 Evaluation

In this section, we will first present the performance measurements of our implementation. We will then perform a comparison of our proposed protocols.

4.6.1 Performance Evaluation

The main goal of our proposal is to enhance the scalability of platform attestation. All proposed protocols have been implemented in Java using the tpm4java framework². The advantage of this framework is that it is a very efficient implementation and we can talk directly to the `/dev/tpm` device driver without requiring that another TPM software stack be present in the system. A software stack in the background would additionally need computation power and thus decrease performance. Our approach causes the time degradation to depend only on the TPM. Further details on the implementation can be found in [63].

We used this implementation to run performance measurements. We measured the runtime of all protocols, excluding those for generating the Diffie-Helman keys and those for generating a new TPM non-migratable key (K_{TS}) required in Protocol 4.4.6. We performed our measurements on a Core2Duo E6700 with 2.66GHz, 2GB RAM running OpenSuSE 10.3 and an Atmel TPM 1.2. The results of our measurements are depicted in Figure 4.4.

The results shown in Figure 4.4 were obtained by averaging over 100 independent runs for each protocol. For each protocol, we measured the latencies when one attestation challenge arrives, as well as when multiple (100) attestation challenges arrive simultaneously. The latencies of the TCG-defined protocol [173] as well as Protocol 3.6.3 scale linearly. Therefore, the time for answering n simultaneously arriving attestation request is approximately equal to $n * x$, where x is the time for answering one attestation.

The average column also depicts the time that is necessary to execute the key confirmation phase once the TPM has delivered the integrity information; this time is shown as the second summand in the column. Note that the TCG-defined IRP in Figure 4.4 does not have a second summand as the TCG-defined protocol does not require a key-establishment.

The time to finish a Multiple-Hash Attestation varies roughly between 1 and 2 seconds, this variation is caused by the ring buffer. If the ring buffer has just been rotated one step and new requests arrive, these requests have to wait until the TPM has finished calculation and the buffer is rotated again. The Tickstamp-Attestation

²<http://tpm4java.datenzone.de>

Protocol	No. concurrent Req.	Average (ms)	Min (ms)	Max (ms)
TCG-IRP [173]	1 n	852.47 $852.47 \cdot n$	842	879
Protocol 3.6.3	1 n	$863.75 + 3.4$ $869.75 \cdot n + 3.4$	860	881
Multiple-Hash Attestation	1 n (Best) n (Worst)	$939.91 + 6.16$ $936.61 + 6.68$ $1826.84 + 6.68$	921 $892 + 1$	973 $1826.84 + 20$
Timestamped Attestation	Token generation time 1 n	1448.95 $<1 + 6.36$ $<1 + 6.36$	1422	1727
Tickstamp Attestation	Token generation time 1 n	1456.58 $<1 + 6.42$ $<1 + 6.42$	1424	1686

Figure 4.4: Measured latencies of selected Integrity Reporting Protocols

and the Timestamp-Attestation also consider the time that is needed to generate the attestation token. As soon as the token has been generated, it can be used to attest to the contents of the platform configuration registers. This token generation time therefore indicates the length of the attestation interval in which this attestation token is used. An attestation interval can, therefore, not be smaller than about 1.5 seconds. The measurements show that all proposed protocols are independent of the number of simultaneously arriving requests. They can therefore significantly reduce the time required to answer simultaneously arriving attestation requests.

4.6.2 Comparison of the protocols

All proposed protocols enable a highly frequented server to timely answer all incoming attestation requests. While the Multiple-Hash Attestation is the slowest and requires roughly between one and two seconds to complete the attestation process, it has the advantage that it is very similar to the existing TCG-proposed integrity reporting protocol. Prover and verifier must, therefore, only minimally modify their attestation interface. This protocol can be classified as an *active attestation*, since it requires the verifier to provide a nonce for the server. To use the protocol a direct connection to the verifier must be established; it is thus not possible to relay the attestation message to other parties.

The biggest advantage of Timestamped Hash-Chain Attestation and the Tickstamp Attestation is that these protocols are *passive attestations*, which require no direct communication between server and client to deliver integrity information. A client can thus collect attestation tokens which he received from other entities and see how the configuration of a particular server changed over time. If this is a desirable feature, the SML must be integrated inside the attestation tokens τ , which would remove the need of a direct connection to the prover and the verifier can collect attestation tokens without noticing the server. However, to ensure freshness of the attestation token and to complete the authentication process, a direct communication between server and client is needed.

The Timestamped Hash-Chain Attestation is based on the TPM's ability to generate a key which is bound to the actual platform configuration. While this is a very elegant solution to realize attestation, its major drawback is that the process of creating one key and binding it to a platform configuration requires a high computational effort. This causes that this process is rather impractical if the platform configuration changes often overtime. We already discussed that issue in Section 3.5.3. However, compared to a client, the platform configuration of a server does not often change overtime. As a result, the concept is especially feasible in the scenario we are considering here.

Both protocols require a TTP either for providing a synchronization token or for the initial nonce. Since the Timestamped Hash-Chain Attestation requires additional security mechanisms, we suggest to use, depending on the scenario, either the Tickstamp Attestation or the Multiple-Hash Attestation.

To prevent an attacker from performing masquerading attacks on the authenticity of the platform configuration, our protocols establish a secure channel between the involved entities. This secure channel is then used for transmitting the privacy-related Stored Measurement Log. Although the computation of these cryptographic operations degrade the server performance, we believe that ensuring the authenticity of integrity information is necessary in order to enable secure integrity reporting.

However, it should also be noted that these cryptographic operations can be computed very efficiently and only increase the time for answering one attestation request by 0.70% and are in the range of milliseconds.

4.7 Summary

In scenarios where an entity is frequently requested to deliver integrity information, existing protocols for performing the TCG-specified platform attestation scale badly. Such scenarios mainly include a client-server architecture, where a large number of clients frequently request integrity information of a particular server. This restriction is caused by the fact that a Trusted Platform Module (TPM) possesses very restricted computation power and is highly involved in the process of platform attestation.

In this chapter, we proposed three solutions to overcome the bottleneck of a TPM and thus to perform platform attestation in scenarios where a frequent integrity verification is needed. Our proposed protocols do not require any modifications to the TPM hardware or any modifications to the measurement process. Although our protocols are based on the TPM-based binary attestation and treat PCR values as measurement data, they could be easily modified to send other forms of measurement data, such as measurement data collected in run-time measurement systems. We presented a performance evaluation as well as a security analysis of our proposed protocol; the results clearly indicate that the protocols can considerably reduce the performance overhead of the attestation process.

Chapter 5

Privacy-Preserving Attestation

Attestation protocols often reveal the platform configuration of the challenger. Since this is not, in every case, a desirable feature, privacy-preserving mechanisms may be necessary in order to be able to make a statement about the trust level of a platform without revealing the exact configuration. In this chapter, we present a mechanism to deal with such a restriction. This chapter shares some material with: *An Approach to a Trustworthy System Architecture using Virtualization* [156] and *Trust, Security and Privacy in VANETs – A Multilayered Security Architecture for C2C-Communication* [159].

5.1 Introduction

One component that is critical for verifying the integrity of a remote platform configuration is the Stored Measurement Log (SML). The SML is a data structure in which all events that affect the platform integrity are stored. The SML is needed for the verifying entity to make a trust decision about the requesting system's platform configuration. However, this approach reveals the platform configuration of the requesting system. Since this is not, in every case, a desirable feature, privacy-preserving mechanisms may be necessary for making a statement about the trust level of a platform without revealing the exact configuration. Without such mechanisms, market dominant vendors could, for example, introduce attestation techniques and deny access to their services if a client runs a competitor's software in parallel. One approach that introduces privacy-preserving mechanisms was proposed by Sadeghi et al. [139]. The authors propose representing a platform configuration not by static integrity metrics, but by a set of properties that characterize a specific application. A webbrowser of two different vendors would, thus, have the same properties. Since simply transferring the properties that characterize a particular application are not privacy related, this approach seems to be a very elegant solution. However, the authors do not specify how a property exactly looks and how a property is generated. Exacerbating this problem is the fact that based on the complexity of current applications, the issue of whether or not small atomic properties that clearly describe a particular application can be extracted

is debatable. In addition, the problem of how to handle recent security vulnerabilities or new software versions is unclear. Assume for example that an application provides a specific property. However, whether this application still provides this specific property when a vulnerability in this application has been found is unclear.

To alleviate these problems, we propose a distributed integrity validation architecture (shown in Figure 5.1). In this approach, integrity validation is performed not through one single entity, but through a number of independent collaborating software integrity validators. This approach enables one to

- outsource the SML validation process so that the producer of a software component verifies the trust level of a software component provided by himself and
- split the SML into self contained subdivided SMLs, where one subdivided SML corresponds to one software component.

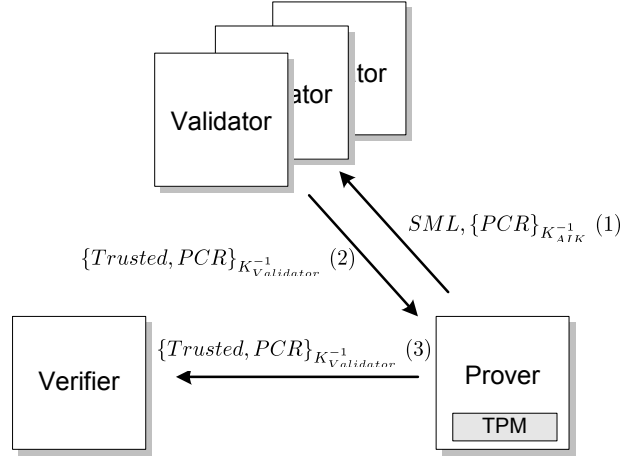


Figure 5.1: Distributed Integrity Validation Architecture

Using this approach, the trust level of a prover can be validated without transferring the complete SML to one single entity. In addition, the structure that describes the platform configuration is split into different parts. Only the possession of all parts of the subdivided SML enable the discovery of the whole platform configuration of a specific entity.

This chapter is organized as follows. We first present the notation used in this chapter in Section 5.2. In Section 5.3, we present the basic idea of our distributed integrity validation architecture and give a high level description of our approach, explaining the different associated algorithms. In Section 5.4, we present our architecture's security protocol, which provides freshness and authenticity of software integrity values. Section 5.5 provides a security analysis of our architecture. Finally, we conclude in Section 5.6.

5.2 Notation

We use the following notation throughout this chapter.

\mathcal{P}	The complete SML as matrix. The matrix has the form
---------------	---

$$\mathcal{P} = \begin{pmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & - \end{pmatrix}. \quad (5.1)$$

Each row of \mathcal{P} represents the measurements performed inside a platform configuration register. For example, m_1, m_2, m_3 are the measurements measured to PCR[0]. A $-$ indicates that no further measurements occurred in this register.

\mathcal{C}	The matrix containing all events that were measured in platform configuration registers. It is a single-column matrix where the first entry denotes the hash-value stored in PCR[0]. The second entry holds the hash value stored in PCR[1] and so on. \mathcal{C} can be generated based on \mathcal{P} by successively hashing each entry in one row of \mathcal{P} by using the classical PCR comutation function (Equation 2.1). \mathcal{C} is typically implemented as a TPM_PCR.COMPOSITE structure in a TPM.
---------------	--

Sig (\mathcal{C})	The signed \mathcal{C} matrix that provides proof of authenticity of software integrity values.
------------------------------	---

n	The platform configuration register number n .
-----	--

i	The i -th measurement event.
-----	--------------------------------

$P_{n,i}$	An entry of \mathcal{P} where n is the platform configuration register number n , i.e., the row-index, and i is the i -the measurement event, i.e., the column-index.
-----------	---

\mathcal{R}	All measurements occurred after $P_{n,i}$. \mathcal{R} is always a single-row matrix.
---------------	--

R_j	One entry of matrix \mathcal{R} . This is simply a hash-value.
-------	--

L	is a hash-value and denotes all measurements <i>before</i> occurrence of the measurement event $P_{n,i}$. It is computed by $L = h(L_{old} i_{next})$.
-----	---

$h(.)$	A one way hash function, i.e., SHA1.
--------	--------------------------------------

\mathcal{M}	is a single-column matrix that holds the signature of the validator on a specific hash-value. One entry of the matrix is referenced with M_j .
---------------	--

T	is a structure where trusted reference values for a specific application are stored. This structure is needed for the validator to verify whether a specific measurement can be validated as trusted.
-----	---

5.3 Ticket-Based Attestation

A SML-structure is basically divided into two parts. The first part reflects the pre-kernel measurement stage (BIOS and TPM-GRUB Boot Loader) while the second part reflects the post-boot measurement stage, as for example, measured by IBM's integrity measurement architecture (IMA) [142]. We performed analysis of the distribution of measurements and the amounts of measurements in both stages. Our analysis shows that a large number of software components are software components coming from one software producer or one software distributor. This is especially true for the pre-kernel measurements, i.e., measurement 1-63, which belong to the BIOS and can be easily verified by the software producer of the BIOS. Thus, it is obvious to combine these measurements into one measurement and represent this combined measurement as a composed hash-value. As a result, the complete SML could be splitted into multiple parts, where one part is represented by a composed hash-value, i.e., the sum of all hash values which can be validated by the same validator. A validator can then validate all measurements used to compose the hash-value and certify the composed hash-value, stating that all measurements are trusted.

In the following, we again call the platform that wants to attest to the contents of its platform register as the *prover*. The platform that receives some sort of measurements and validates the trust level of the prover is referred to as the *validator*. The validator issues some sort of *ticket* that states that a specific measurement or a set of measurements is trusted. This ticket can then be transferred to a *verifier* to prove that the platform of the prover is trusted.

5.3.1 High-Level Description

The hierarchy of our proposed SML structure is shown in Figure 5.2. The complete SML, which clearly describes the integrity of a platform, is composed of an implementation specific number of subdivided SMLs where each sub-SML describes all software components measured in one particular platform configuration register. For each SML, there again exists one atomic SML-entry that is related to each software integrity metric and measured in one PCR. We refer to this atomic measurement entry as a measurement event.

To validate the trust level of a specific platform, the prover transmits all measurement events of a specific PCR, together with the hash-value of the PCR to the entity that can make a trust decision for the measurement related software component, i.e., the software producer or some sort of software validator. The deciding entity then compares the measurement with trusted reference values that he manages and verifies whether this measurement was used to calculate the received value of the PCR. If the verification succeeds, the validator creates a special purpose data structure that vouches for the trust level of the software measurement. This special purpose data structure must be cryptographically protected and bound to the platform that provided the platform integrity values in order to ensure freshness, authenticity and confidentiality. For the sake of simplicity, we view this structure as a signature created by the validator

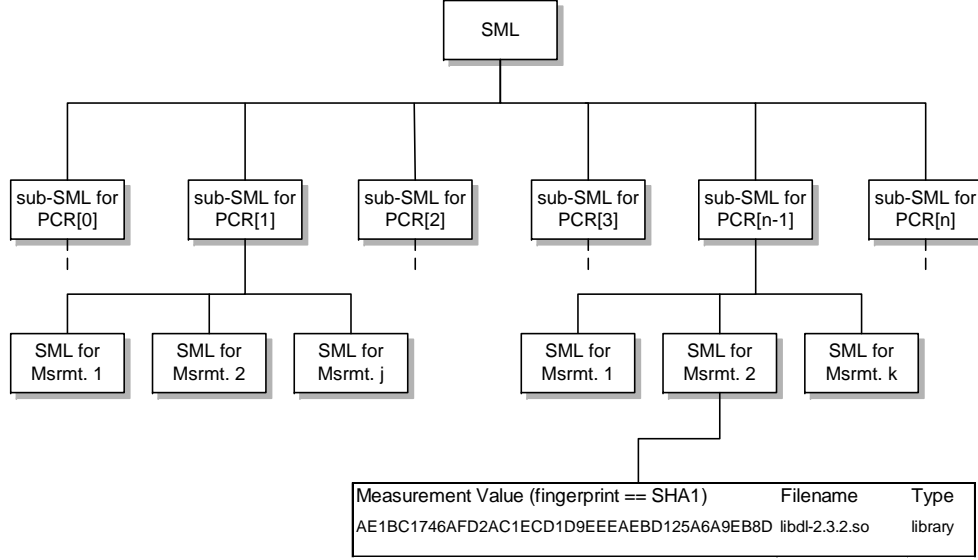


Figure 5.2: Hierarchy of the SML

on a specific measurement event and refer to it as *ticket*. We will discuss the detailed syntax of this structure and the security mechanisms in Section 5.4. If a special purpose data structure can be obtained for every measurement, a verifier can validate that the platform configuration is trusted.

5.3.2 Integrity Reporting Algorithm

In the following, we present in detail how the proposed concept of Section 5.3 can be realized. For this purpose, we assume that a system wants to report to another entity that it is in a trusted system configuration without revealing the complete system configuration to the verifier. A measurement engine runs on the system of the prover and has generated the matrix \mathcal{P} (which is simply the SML). The prover then wants to receive from the validators a ticket that proves that his system is in a trusted state. To prevent that each validator receives the complete matrix \mathcal{P} (which would reveal the complete system configuration of the prover), the prover compresses the hash values, so that a specific validator only receives the part of \mathcal{P} that he needs to receive in order to make a trust decision. For this purpose, the prover compresses the measurements and generates L .

L is generated as follows. All measurements $(1, \dots, i-1)$ that occurred before measurement event number i can be compressed into one hash-value by iteratively computing the function $L = h(L_{old} || i-1)$ with $i > 2$. This process increases privacy since a validator cannot compute the pre-image of h_{i-1} so as to determine the exact measurement event $i-1$. However, since the used SHA1 hash-function h is not homomorph and non-associative under the concatenation operation, i.e., $h(a) || h(b) \neq$

$h(a || b)$ and $h(a || b) || c \neq h(a || (b || c))$, we cannot analogously compress \mathcal{R} into one hash-value. Before we present the algorithm that realizes this description, we will first explain the basic scheme using a simple example.

Suppose the following 160-bit SHA1 measurements m_1, m_2, m_3, m_4, m_5 were measured on a system. Measurements m_1 to m_3 have been measured into PCR[0] and m_4, m_5 have been measured into PCR[1], i.e., PCR[1] holds the value $k = h(m_4 || m_5)$. Thus, the matrix \mathcal{P} is of the following form:

$$\mathcal{P} = \begin{pmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & - \end{pmatrix} \quad (5.2)$$

For each entry in matrix \mathcal{P} , the matrix \mathcal{R} and the hash-value L are computed. The scheme starts with the first entry of PCR[0]. Thus, $P_{0,0} = m_1$, $L = ()$, indicating an empty matrix, and $\mathcal{R} = \begin{pmatrix} m_2 & m_3 \end{pmatrix}$. The hash-value $P_{0,0} = m_1$, \mathcal{R} , L , and **Sig**(\mathcal{C}) are transferred to a specific validator. If $P_{0,0}$ is trusted, the validator delivers a signature on m_1 back to the prover. The second measurement $P_{0,1} = m_2$ yields $L = m_1$ and $\mathcal{R} = \begin{pmatrix} m_3 \end{pmatrix}$. $P_{0,1}$, \mathcal{R} , L , **Sig**(\mathcal{C}), and the signature on m_1 are transferred to a specific validator. If $P_{0,1}$ is trusted, the validator delivers a signature on $h(m_1 || m_2)$ back to the prover. The third measurement $P_{0,2} = m_3$ yields $L = h(m_1 || m_2)$ and $\mathcal{R} = ()$. The hash-value $P_{2,0} = m_3$, \mathcal{R} , L , **Sig**(\mathcal{C}) and the signature on $h(m_1 || m_2)$ are again transferred to a validator. If $P_{0,2}$ is trusted, the validator delivers a signature on $h(h(m_1 || m_2) || m_3)$ back to the validator and this signature is placed in M_0 . The algorithm continues with $P_{1,0} = m_4$ and keeps on running until all measurements are transferred to a validator.

The algorithm for reporting the integrity of a prover to a certain validator is shown in Algorithm 1. This algorithm processes each recorded measurement event and transfers the corresponding measurement event $P_{n,i}$, the hash-value of all previous measurements L , a signature of all previous measurements **Sig**(L), all following measurements \mathcal{R} and **Sig**(\mathcal{C}) to a validator.

The algorithm requires the SML matrix \mathcal{P} , **Sig**(\mathcal{C}), and the number of platform configuration registers a , i.e., (PCR[0, ..., a]), as input. As output, the algorithm returns the matrix \mathcal{M} , where each entry of matrix \mathcal{M} holds a signature of a 160-bit hash value. The algorithm returns (RET_FAIL) if the validation failed and (RET_SUCCESS) if the validation was successful.

Algorithm 1 uses the functions **getMeasurement**($P_{n,k}$), which simply returns the 160-bit measurement of measurement event $P_{n,k}$, and **getValidator**($P_{n,i}$), which returns the unique identifier of the software producer or the validator of measurement event $P_{n,i}$. This process is analogous to the TCG-defined SML verification and can, for example, be accomplished by a public database that includes information about the validator and their supplied software. The function **getNumberOfSMLs**(\mathcal{P}, n) returns the number of measurement events that were recorded into PCR[n].

Algorithm 1: Algorithm for Integrity Reporting

```

Input  :  $\mathcal{P}, \mathbf{Sig}(\mathcal{C}), a$ 
Output:  $\text{RET\_SUCCESS} + \mathcal{M} \mid \text{RET\_FAIL} + P_{n,i}$ 
 $\mathcal{M} := 0$ 
for  $n := 0$  to  $a$  do
   $L, \mathbf{Sig}(L) := 0$ 
   $\mathcal{R} := 0$ 
   $i := 0$ 
  while  $i \leq \text{getNumberOfSMLs}(\mathcal{P}, n)$  do
    for  $j := i + 1$  to  $\text{getNumberOfSMLs}(\mathcal{P}, n)$  do
       $R_j := \text{getMeasurement}(P_{n,j})$ 
     $V_{n,i} := \text{getValidator}(P_{n,i})$ 
    Transmit  $(\mathcal{P}_{n,i}, n, i, \mathcal{R}, L, \mathbf{Sig}(\mathcal{C}), \mathbf{Sig}(L))$  to  $V_{n,i}$ 
    Wait until  $V_{n,i}$  replies
    if  $\text{reply} \neq \mathbf{Sig}(L)$  or  $\text{reply} = \text{"not trusted"}$  then
       $\text{return } (\text{RET\_FAIL}; P_{n,i})$ 
    if  $L = 0$  then
       $L := h(\text{getMeasurement}(P_{n,i}))$ 
    else
       $L := h(L \parallel \text{getMeasurement}(P_{n,i}))$ 
     $i++$ 
   $\mathcal{R} := 0$ 
   $M_n := \mathbf{Sig}(L)$ 
return  $(\text{RET\_SUCCESS}; \mathcal{M})$ 

```

5.3.3 Integrity Validation Algorithm

The algorithm for integrity validation is executed by the validator. The algorithm verifies whether:

- the reported software measurement is consistent with one trusted reference software integrity measurement and thus included in T ,
- the signature of $\mathbf{Sig}(\mathcal{C})$ is valid, stating that $\mathbf{Sig}(\mathcal{C})$ was created by an authentic TPM,
- the signature of $\mathbf{Sig}(L)$, which was created by the previous validator is valid, stating that all software components that were measured *before* $P_{n,i}$ are trusted, and
- the reported software measurement is a component used to generate $\mathbf{Sig}(\mathcal{C})$ and, thus, part of the platform configuration of the prover.

The integrity validation algorithm requires the output of Algorithm 1 as well as a set of trusted reference values T as input. It creates a special purpose signature structure

$\mathbf{Sig}(L)$, which states that the measurement event $P_{n,i}$ and all previous measurements are trusted. The detailed syntax and the security mechanisms of the structure $\mathbf{Sig}(L)$ are discussed in Section 5.4. Before we present the algorithm, we again explain the basic scheme using an example. For this purpose, we continue the example from above.

Suppose $P_{0,0} = m_1, \mathcal{R}, L$ and $\mathbf{Sig}(\mathcal{C})$ have been received by a specific validator. The validator first checks that m_1 is included in T and, thus, is consistent with a trusted reference software integrity measurement. The validator then verifies whether $\mathbf{Sig}(\mathcal{C})$ was created on an authentic TPM by verifying $\mathbf{Sig}(\mathcal{C})$ and whether $\mathbf{Sig}(L)$ was created by a known validator. Then, the validator tries to recompute \mathcal{C} in order to check that $P_{0,0}$ is part of the signature. For this purpose, the validator computes $V = h(h(m_1 || m_2) || m_3)$ and uses the public-key of the prover to verify that V is equal to \mathcal{C} . If everything checks out, the validator computes a signature on m_1 by computing $L = h(L || m_1)$ and signing L with his private key. The signature on L , $\mathbf{Sig}(L)$ is then transferred back to the prover.

Algorithm 2: Algorithm for Integrity Validation

```

Input  :  $P_{n,i}, i, \mathcal{R}, L, \mathbf{Sig}(L) \mathbf{Sig}(\mathcal{C}), T$ 
Output:  $\text{RET\_FAIL} + P_{n,i} \mid \text{RET\_SUCCESS} + \mathbf{Sig}(L)$ 
if  $\text{getMeasurement}(P_{n,i}) \in T$  then
    if  $\text{VerifySig}(\mathbf{Sig}(\mathcal{C}))$  and  $\text{VerifySig}(\mathbf{Sig}(L))$  then
         $V := h(L || \text{getMeasurement}(P_{n,i}))$ 
         $\mathcal{L} := h(\mathcal{L} || \text{getMeasurement}(P_{n,i}))$ 
        for  $j = i + 1$  to  $\text{MaxNumber}(\mathcal{R})$  do
             $V := h(V || R_j)$ 
        if  $V = \text{getPCRMeasurement}(\mathbf{Sig}(\mathcal{C}), n)$  then
            Create  $\mathbf{Sig}(L)$ 
            return  $\text{RET\_SUCCESS}; \mathbf{Sig}(L)$ 
    return  $\text{RET\_FAIL}; P_{n,i}$ 

```

The algorithm for integrity validation is shown in Algorithm 2. It uses the function $\text{VerifySig}(\mathbf{Sig}(\mathcal{C}))$ and $\text{VerifySig}(\mathbf{Sig}(L))$. Both simply verify that the signature of $\mathbf{Sig}(\mathcal{C})$, and the signature of all previous measurements are valid. It returns RET_SUCCESS if all checks succeed. The function $\text{MaxNumber}(\mathcal{R})$ returns the number of measurement events in \mathcal{R} and $\text{getPCRMeasurement}(\mathbf{Sig}(\mathcal{C}), n)$ returns platform configuration register n 's 160-bit hash value used to compute the signature over all platform configuration registers.

5.3.4 Performance Optimizations

Our proposed algorithms work well if the trust level of a system is represented by a small number of measurements. If the system is characterized by a large number of software components running on the machine, such as in a complex open and non-resource constraint operating system environment, the process gets too complex and

is, thus, impractical. In addition, it seems unlikely and impractical that the validator of the measurement m_i is able and willing to validate the signature of the previous measurement m_{i-1} ; thus, certifying that all software components measured before m_i are trusted.

To overcome these restrictions and to provide performance optimizations, we assume that each platform configuration register is validated by one validator. This is a legitimate assumption, since future next-generation secure operating system environments, that are either based on a virtual machine monitor [67, 20, 156] or on a microkernel [68, 137, 101] provide better isolation and also measured execution environments. In particular, these secure operating system environments enable establishing a Dynamic Root of Trust for Measurement (DRTM) [38, 6, 92]. A DRTM effectively removes the measurements of the BIOS, ROM and Bootloader from the trust chain. It is, thus, possible to establish a new trust chain with the DRTM as trust anchor. Additionally, using a DRTM significantly reduces the amount of measurements necessary to clearly describe the state of a platform, thereby causing the trust level of a specific software component to not only depend on those software components running on the platform, but only on software components that are part of the software component's execution environment.

For optimizing the performance, we divide the integrity validation into intermediate hash-computation steps that are certified by only one validator. Since intermediate steps are only represented by one hash-value, a verifier can hardly determine the exact platform configuration. Our division is possible because all measurements by an integrity measurement architecture are performed iteratively and represented by a hash-chain. This means, in effect, that the value of each platform configuration register is validated by a single validator, which causes that only one signature operation per platform configuration register needs to be performed.

One verifier is responsible for one intermediate step and in order to verify the signature of a $\mathbf{Sig}(\mathcal{C})$, all intermediate steps must have a signature. The prover then transmits to the verifier the signed special purpose data structures for each intermediate step as well as $\mathbf{Sig}(\mathcal{C})$. The verifier then verifies the trust level of the prover by re-composing \mathcal{C} based on the information given in the intermediate attestation steps and by validating $\mathbf{Sig}(\mathcal{C})$.

Each entry of matrix \mathcal{P} is again denoted $P_{n,i}$ and P_n denotes the n -th row vector in \mathcal{P} , which are actually all software integrity measurement events measured into $\mathbf{PCR}[n]$. For simplicity reasons, we assume that a measurement event is only a SHA1 fingerprint and does not include information about the filename, the type of file and so on. A platform configuration is divided into a number of intermediate steps, which are denoted IS_n , with $n = 1, \dots, x$ where each intermediate step is represented as a SHA1 hash-value. One intermediate step, IS_n , is composed by computing Equation (5.3) and is a hash-value over all entries in P_n :

$$IS_n = h(\dots h(h(P_{n,0} \parallel P_{n,1}) \parallel P_{n,2}) \dots). \quad (5.3)$$

All P_n are transferred to their respective validator V_n using Algorithm 3, who computes Equation 5.3 and creates a special purpose signature on IS_n . We denote this structure as $\mathbf{Sig}(IS_n)$. We again explain this scheme using a simple example.

Suppose we have the 160-bit SHA1 measurements m_1, m_2, m_3, m_4, m_5 . Measurements m_1 to m_3 have been measured into PCR[0] and m_4, m_5 have been measured into PCR[1]. Thus, the 1-st row vector in matrix \mathcal{P} is of the following form:

$$P_1 = (m_1 \quad m_2 \quad m_3) \quad (5.4)$$

For each row vector in matrix \mathcal{P} , the matrix P_n and $\mathbf{Sig}(\mathcal{C})$ are transferred to the validator. For P_1 the validator delivers a signature on the first intermediate step back. This signature is simply a signature on $h(h(m_1 || m_2) || m_3)$. The algorithm continues with $P_2 = (m_4 \quad m_5 \quad -)$.

In the following, we only present the algorithm for integrity reporting with intermediate steps as shown in Algorithm 3. The algorithm for integrity verification of intermediate steps can be constructed analogously and is rather similar to Algorithm 1. The only difference is that validating the signatures of the previous validator can be skipped.

Algorithm 3: Algorithm for Integrity Reporting with intermediate steps

Input : $\mathcal{P}, \mathbf{Sig}(\mathcal{C}), a$
Output: RET_SUCCESS + \mathcal{M} | RET_FAIL + $P_{n,i}$
 $\mathcal{M} := 0$
for $n := 0$ **to** a **do**
 $V_n := \text{getValidator}(P_n)$
 Transmit $(P_n, \mathbf{Sig}(\mathcal{C}))$ to V_n
 Wait until V_n replies
 if $\text{reply} \neq \mathbf{Sig}(IS_n)$ **or** $\text{reply} = \text{"not trusted"}$ **then**
 return ("Validation failed on"; $P_{n,i}$)
 $M_n := \mathbf{Sig}(IS_n)$.
return (RET_SUCCESS; \mathcal{M})

5.3.5 Integrity Verification

If the prover's platform is trusted, he has received, for each intermediate step, a signature of the form $\mathbf{Sig}(IS_n)$. To prove to a certain verifier that his platform has received attestation and is, thus, trusted, he transfers the matrix \mathcal{M} together with $\mathbf{Sig}(\mathcal{C})$ to the verifier. The verifier verifies that all signatures of $\mathbf{Sig}(IS_n)$ with $n = 1, \dots, a$ are valid and that Equation 5.5 and 5.6 holds.

$$t = h(IS_0 || IS_1 || IS_2 || \dots || IS_n) \quad (5.5)$$

$$t \stackrel{?}{=} E(K_{AIK}, \mathbf{Sig}(\mathcal{C})) \quad (5.6)$$

The verification can be computed using Equation (5.6) because $\mathbf{Sig}(\mathcal{C})$ is a signed TPM_PCR_COMPOSITE structure [168] which is computed by simply concatenating all hash-values of the platform configuration and then computing the signature of the hash-value ([168], pp. 69). Using Equation (5.6) for the verification computation provides privacy to the platform configuration of the prover because his platform configuration is only represented by hash-values and no SML needs to be transferred.

5.4 Security Mechanisms

In the last sections, we presented the basic concept of dividing a complete SML into smaller self-contained parts and showed how this concept can be used to provide a proof of the trust level of a system. We also presented the algorithms that can be used to realize this concept. However, this concept requires security mechanisms and a special type of attestation. In this section, we present the necessary security mechanisms that enable using our proposed concept. The protocol proposed herein is based on the same foundation as Protocol 4.4.6, which we already presented in Chapter 4. Thus, it satisfies Requirement 1 and Requirement 3 of a secure attestation channel. For simplicity reasons, we do only present the main protocol steps without negotiation of an additional symmetric key between verifier and prover. However, the protocol can be easily extended with the necessary steps of negotiating a key (e.g., by adapting steps (1) - (4) of Protocol 4.4.6 into the protocol). The integration of the necessary steps is trivial and, hence, omitted here. As a result, Requirement 2 is satisfied if an additional key-establishment is integrated inside the protocol. Under that circumstances, an attacker cannot eavesdrop on the communication channel and learn P .

The ticket-based attestation protocol is shown in Protocol 5.4.1. The protocol only provides one possible realization of the concepts introduced in this chapter. Since the creation of a TPM-bound key might result in performance degradation (Compare Section 3.5.3), the security mechanism could also be adapted depending on the application scenario. We will, for example, use a modified version (adapted to e-commerce) of this protocol in Chapter 9.

Protocol 5.4.1: Ticket-based Attestation

SUMMARY: Secure integrity reporting with integrity verification and ticket issuing
 RESULT: Ticket-based attestation: key-establishment with key-confirmation

1. *Notation.*

V_n denotes a validator with $n \in \{1, \dots, a\}$

B is the prover

A denotes a verifier

P_n denotes the n -th row of matrix \mathcal{P} , i.e., a part of the SML

2. *System Setup.*

V_x and A must acquire (and validate) the certificate of the Privacy-CA to validate

$Cert(AIK, K_{AIK})$

B must acquire (and validate) the certificates of V_n to verify $\mathbf{Sig}(IS_n)$

3. *Protocol messages of the certification phase.*

$$V_n \leftarrow B : Cert(AIK, K_{AIK}), P_n, Cert(TK, K_{TK}) \quad (I)$$

$$V_n \rightarrow B : \{\mathbf{Sig}(IS_n)\}_{K_{TK}} \quad (II)$$

4. *Protocol messages of the attestation phase.*

$$A \leftarrow B : \mathcal{M}, Cert(AIK, K_{AIK}), Cert(TK, K_{TK}) \quad (1)$$

$$A \rightarrow B : \{Na, K_{AB}\}_{K_{TK}} \quad (2)$$

$$A \leftarrow B : \{Na\}_{K_{AB}} \quad (3)$$

5. *Protocol actions.*

- (a) *Precomputation and Pre-deployment by B.* B creates a non-migratable TPM key (K_{TK}) that is bound to a specific set of platform configuration registers. B certifies K_{TK} with K_{AIK}^{-1} . The resulting structure is denoted $Cert(TK, K_{TS})$ and includes a `TPM_PCR_INFO` structure which gives an assertion to which PCRs K_{TK} is bound. The AIK signed `TPM_PCR_COMPOSITE` structure $\mathbf{Sig}(C)$ is also part of this certificate.
- (b) B executes Algorithm 3 which transfers for each platform configuration register, the corresponding P_n in message (I) to the verifier responsible for validating P_n .
- (c) V_n verifies whether the intermediate step is trusted using Algorithm 2. For this purpose, V_n verifies all signatures and validates whether all entries of P_n are trusted. In addition, B verifies whether he can re-construct the C used to generate $\mathbf{Sig}(C)$ using Equation (5.1).
- (d) V_n computes Equation 5.1 and 5.2 and transfers message (II) to B .

$$IS_n = h(\dots h(h(P_{n,0} || P_{n,1}) || P_{n,2}) \dots), \quad (5.1)$$

$$\mathbf{Sig}(IS_n) = \{timestamp, Certifier, IS_n, K_{TK}\}_{K_{V_n}^{-1}}. \quad (5.2)$$

B has now received a set of $\mathbf{Sig}(IS_n)$ which is denoted \mathcal{M} and can subsequently be used by B to prove that he obtained attestation. After completing the certification phase, B is ready to answer attestation requests.

- (a) To prove to A that B has obtained attestation, he transfers message (1) to A .
- (b) A verifies whether he can re-construct a `TPM_PCR_INFO` structure using the information transferred in \mathcal{M} . If the re-constructed `TPM_PCR_INFO` structure is consistent to the one transmitted in $Cert(TK, K_{TK})$ and the signature validation of $Cert(TK, K_{TK})$ succeeds, the matrix \mathcal{M} is untampered and

all measurements of \mathcal{M} are also reflected inside the PCRs of the TPM. A then verifies all signatures and that he trusts the validator V_n to certify $V_n, \forall n \in \{1 \dots a\}$. In addition, B verifies that the key K_{TK} is a TPM bound key which is bound to the platform configuration registers certified by V_n . If all checks succeed, A creates a non-predictable nonce Na and a session key K_{AB} . B then transfers message (2) to B .

- (c) B is only able to decrypt message (2) if his current platform configuration still matches the platform configuration when the key K_{TK} was first generated. B decrypts message (2) and uses K_{AB} to transfer message (3) to A .
- (d) A verifies the freshness of Na . If this holds, B has provided a proof that he obtained attestation and that his current platform configuration is consistent to the platform configuration certified by $V_n, \forall n \in \{1, \dots, a\}$.

5.5 Security Analysis

In this section, we perform an analysis of the security mechanisms of Protocol 5.4.1. The security of Protocol 5.4.1 relies on the assurance that a specific non-migratable TPM key, satisfying certain criteria, is used. This assurance is made using a certificate generated through `TPM.CertifyKey`. It has already been verified by Löhner et al. [103], Sadeghi et al. [137] and Stumpf et al. [160] that this concept enables secure integrity reporting. We will, thus, only verify that our proposal provides the same security as [103, 137, 160].

5.5.1 Certification Phase

This section analyzes whether our proposed concept can be used to validate the trust level of a complete system.

The state of a TPM-enhanced platform is reflected by the values of a set of platform configuration registers. To transmit a set of platform configuration registers and to attest to their contents, a signed structure **Sig**(\mathcal{C}), of the form **Sig**(\mathcal{C}) = $\{\text{PCR}[0] \parallel \text{PCR}[1] \parallel \text{PCR}[2] \parallel \text{PCR}[3] \parallel \dots\}_{K_{AIK}^{-1}}$ is created. We again assume, for simplicity reasons, that $P_{n,i}$ denotes the i -th 160-bit measurement measured in platform configuration register n . Each PCR holds a 160-bit hash value, which is composed of a hash-value of the form $h_{P_{n,i}} = h(h_{P_{n,i-1}} \parallel P_{n,i})$. Each validator of a PCR receives P_n , which is the n -th row of matrix \mathcal{P} . A validator verifies that the received P_n is trusted by comparing it with reference values. He can then construct \mathcal{C} , by computing Equation (5.1), and verify that Equations (5.2) holds. Therefore, **Sig**(\mathcal{C}), can be rewritten as $S = \{h(h(\dots h(h(P_{0,0} \parallel P_{0,1}) \parallel P_{0,2}) \dots) \parallel \text{PCR}[1] \parallel \text{PCR}[2] \parallel \text{PCR}[3] \parallel \dots)\}_{K_{AIK}^{-1}}$. Each validator of a PCR receives P_n which is the n -th row of matrix \mathcal{P} .

$$\mathcal{C} = h(\text{PCR}[n-1] \parallel h(\dots h(h(P_{n,i-1} \parallel P_{n,i}) \parallel P_{n,i+1}) \dots) \parallel \text{PCR}[n+1] \dots), \quad (5.1)$$

$$\mathcal{C} \stackrel{?}{=} E(K_{AIK}, \mathbf{Sig}(\mathcal{C})). \quad (5.2)$$

If, for every verifier n , all $P_{n,i}$ are trusted, and for each verifier, Equation 5.2 holds, the complete system is trusted. This holds since $E(K_{AIK}, \mathbf{Sig}(\mathcal{C})) = h(\dots h(h(\dots h(h(P_{n,i-1} || P_{n,i} || P_{n,i+1}) \dots) || h(h(P_{n+1,i-1} || P_{n+1,i} || P_{n+1,i+1}) \dots) || \dots))$. Each verifier creates a ticket of the form $\{timestamp, Certifier, IS_n, K_{TK}\}_{K_{V_n}^{-1}}$, which certifies a specific intermediate step, i.e., PCR, as trusted. Only the possession of all tickets enables a clear description of the state of a platform. If a ticket is missing or one or more tickets are issued for a key other than K_{TK} , the complete platform is in an untrusted state. Replaying an old ticket or relaying it to another platform is not possible, since the ticket is bound to a specific key K_{TK} and, thus, only usable on the platform where K_{TK} is located. Since this key is also protected by a hardware TPM, it is hard for an attacker to extract this key.

5.5.2 Attestation Phase

To enable proving that a platform configuration is trusted, the prover transfers the matrix \mathcal{M} , which contains all $\mathbf{Sig}(IS_n)$ with $n \in \{1, \dots, a\}$ to the verifier. The verifier validates that all entries have a valid signature and that he trusts the certifier to create signatures on a specific hash-value. It is thus important that there exists a public-key infrastructure, that makes a certifier liable for their issued tickets. Otherwise, it would be possible for a malicious certifier to certify unknown or malicious hash-values. However, it should be noted that this problem also arises within the TCG-defined SML validation. The TCG-defined SML validation requires so-called Reference Integrity Metric (RIM) Certificates [172] which are signed by a trusted certifier and are comparable to the ticket we introduced in Section 5.2. Therefore the TCG-defined SML validation process does not provide better security or liability than our approach.

A general goal for an attacker in the process of platform attestation is to try to replay an unfresh platform configuration, so as to masquerade an untrusted platform configuration as trusted. For this purpose, the attacker could try to generate multiple keys, K_{TK} , which are bound to a broad, diverse set of non-trusted platform configurations. He could then execute the certification phase of Protocol 5.4.1, trying to collect as many $\mathbf{Sig}(IS_n)$ as possible, with all keys he generated previously. Afterwards, he tries to compose a complete \mathcal{M} using the answers of the validators. However, since for each key only a subset of IS_n is trusted, he would not be able to compose a complete \mathcal{M} belonging to only one specific K_{TK} . Thus, if he were to use his composed \mathcal{M} to prove to a certain verifier that he obtained attestation, the validator could detect this attack since not all tickets have been issued for the same key K_{TK} .

Another attack could involve a verifier creating a bound key K_{TK} in a non-trusted platform configuration. He could then collect and store all $\mathbf{Sig}(IS_n)$ that he is able to obtain under this platform configuration. Note that he would not get a certificate for each IS_n , since his platform is not trusted. The attacker could then change into an untrusted platform configuration where the missing IS is trusted. He would then try to

re-generate the same key K_{TK} but under the new platform configuration. This would enable him to compose a complete \mathcal{M} , which would state that his complete platform configuration is trusted. However, this attack fails, since the attacker has to use the true random number generator of the TPM and it is computationally infeasible to again compute the same key K_{TK} .

Replay, relay, or man-in-the-middle attacks on the ticket $\mathbf{Sig}(IS_n)$ are not successful, since the ticket is bound to a specific key K_{TK} . Attacks of this type are detected by the challenge-response phase accomplished in steps (2) and (3). If the verifier is not trusted for generating the session key K_{AB} , Protocol 5.4.1 can be easily extended by using a Diffie-Helman-based key-establishment as, for example, proposed in Chapter 3.

Our proposed solution does not reveal the exact platform configuration to a verifier, because the trust-level of a platform configuration is represented by a set of signed hash-values. Since each hash-value is computed based on Equation (5.1) and the used hash-function is pre-image resistant, it is computationally infeasible to find the input parameters of the pre-image. However, the signature of the validator permits drawing conclusions concerning the software application running on the prover's platform. If, for example, only one validator is responsible for ticket issuance of one software application and, thus, one measurement event, a validator can rediscover that this specific application is running on the prover's platform. To alleviate this problem, blinded signatures combined with zero-knowledge proofs [85, 28, 23] could be used. This approach would enable each validator to obtain a certificate on a blinded public key and, thus, to embed the public key into a public key hierarchy. The verifier could then validate that a specific public key is part of a key hierarchy without directly recovering the identity of the validator. However, the exact specification of such a cryptographic certification protocol is not part of this thesis and remains for future work.

5.6 Summary

In this chapter, we proposed a distributed integrity validation architecture that allows outsourcing the attestation process. Our approach enables dividing a whole platform configuration into self-contained, independent parts. All independent parts are transferred to independent validators in order to verify the trust level of one self-contained part. Since all independent parts can be combined to represent the whole platform configuration, a platform can prove its complete trust level to a second entity by transferring all independent and validated parts. For our proposed concept, we also presented a security protocol that provides freshness and authenticity of software integrity values and prevents an adversary from masquerading an untrusted platform configuration. Our proposed protocol is secure, since it is based on the property of a non-migratable key bound to a specific platform configuration. While our protocol is rather impractical if it is used in very complex non-constraint operating systems characterized by a large amount of measurements, it is utilizable in next-generation operating system environments based on virtualization or in embedded systems. In these environments,

the trust level of a platform can be represented by a small number of measurements, thereby making our proposed solution to be reasonable practical.

Chapter 6

Lightweight Attestation

In this chapter, we show how attestation techniques can be realized in systems with very low computation power. This chapter shares some material with: *Detecting Node Compromise in Hybrid Wireless Sensor Networks using Attestation Techniques* [96].

6.1 Introduction

In the preceding chapters, we have seen how the trusted platform module can be used to establish secure attestation channels. The protocols proposed therein require asymmetric cryptography and a relative high computation power to compute and establish shared keys, which enable establishing an authentic attestation channel. In addition, these protocols have an intrinsic complexity as some sort of validation entity has to determine the trust level of a particular communication partner using the complex SML. We referred to this process as *explicit attestation* (compare Section 3.2). The explicit attestation may be very complex, especially when many software components are involved and are represented by the SML. To make a statement of the trust level of a certain platform, it is necessary that the manufacturer of a particular software provides trusted reference values, with which the SML can be processed and the resulting values be re-computed. If one manufacturer is not providing reference values for a particular process, or the SML contains a process to which no trusted reference value can be found, the complete software system should be declared as being not trusted. Thus, this flexibility of being able to report a big amount of measurements is also its major drawback. However, in resource-constrained systems or systems with very low computation power, such as wireless sensor networks or embedded systems, a complex attestation process that is based on asymmetric cryptography is impractical. With respect to attestation techniques, these systems are characterized by two issues:

1. These systems do often not possess enough computation power to perform asymmetric cryptography.
2. The software configuration of these systems often does not change during their whole lifetime.

In this chapter, we propose two lightweight TPM-based attestation protocols for resource-constrained systems. These protocols allow performing an *implicit attestation* by utilizing the fact that the software configuration of resource-constrained systems often does not change during their whole lifetime. Our proposed protocols enable a low-cost node to verify the trust level of another node which possesses more computation power and which are equipped with a Trusted Platform Module. Both protocols do not require expensive public key cryptography and the exchanged messages are very short.

This chapter is organized as follows. In Section 6.2, we introduce the setting of our work. In this context, we present two application scenarios for lightweight attestation techniques and show for each scenario that there is a need for using attestation techniques. In Section 6.3, we show the assumptions of our work and also explain the specific notation we are using in this chapter. Section 6.4 is the core part of this chapter and deals with the attestation protocols, which we propose for securing resource-constrained systems. In Section 6.5, we analyze our proposed protocols regarding security and performance and in Section 6.7, we finally present a summary of this chapter.

6.2 Setting

Due to the large scale and desired low-cost of embedded networks, it is not feasible to integrate a TPM into each individual node. Fortunately, many embedded networks are organized in clusters where a minority of nodes perform some special functions. We consider such a cluster-organized network. It consists of two different types of nodes. One type of nodes are low-cost cluster nodes (CNs) and the other type of nodes are more expensive super nodes (SNs) which are additionally equipped with a TPM. TPM-equipped nodes act as super node for a number of low-cost cluster nodes. The super nodes possess more computation power and perform a number of special operations such as data aggregation or key-management for the cluster nodes in their vicinity.

In the following, we present two different scenarios for this setting. We will also show for each scenario, why there is a need for applying attestation techniques and which threats evolve in these scenarios.

6.2.1 Scenario: Wireless Sensor Networks

A perfect example for a network that can be organized in clusters are Wireless Sensor Networks (WSNs) [2]. These networks provide a technological basis for many different security-critical applications, such as military surveillance, critical infrastructure protection and surveillance. WSNs can be deployed in unattended and even hostile environments for monitoring the physical world. The monitored environment is covered by hundreds or even thousands of sensor nodes with embedded sensing, computation, and wireless communication capabilities. If sensor nodes are not specially protected, an adversary can easily compromise them, recover information (e.g., keying material) stored on the nodes, and subvert them to act as authorized nodes in the network to perform insider attacks.

Figure 6.1 shows a cluster of a wireless sensor network. The figure shows a number of different cluster nodes, realized by the Berkeley Mica Motes [41] as they are communicating with one Stargate platform [42] acting as super node.

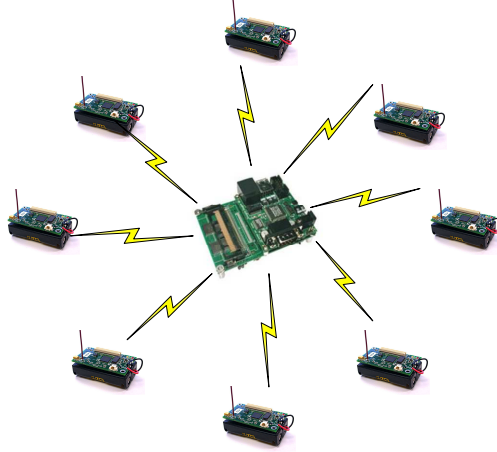


Figure 6.1: Cluster of a Wireless Sensor Network

Especially a super node is a valuable target for an adversary since these nodes perform some special operations and also possess all cryptographic keys of all sensor nodes in the vicinity. If an adversary is able to compromise one super node, thus, getting access to all stored cryptographic keys, he can easily frame all sensor nodes in his vicinity and also easily inject bogus data to the network. If an adversary would have compromised only one cluster node, injecting false reports into the network would be much more difficult, as the super node may verify whether other nodes also report the same event before he transmits the report to the sink. An adversary can therefore inflict substantial damage to the whole network, by only compromising one super node.

A sensor network consisting of super nodes and cluster nodes can be deployed randomly, e.g., via aerial scattering. That means the immediate neighboring nodes of any sensor node are not known in advance. Super nodes and cluster nodes communicate with each other through a multihop communication, which is caused amongst others by the limited power supply of each node. The sensed data is sent via multihop communication to the *sink*. The sink is assumed not to be constrained in its resources and cannot be compromised. It possesses all keying material shared with the sensor nodes.

6.2.2 Scenario: Embedded Systems

Besides wireless sensor networks, cluster-organized networks are increasingly being used in the realm of embedded systems and processors, where a number of nodes with small computation power communicate with other embedded controllers that possess more computation power. Since embedded controllers are more and more integrated in nearly every electronic device. Examples can be found in the whole transportation systems from planes to vehicles. Automotive safety systems such as the anti-lock braking system

(ABS), Electronic Stability Control (ESC/ESP), and automatic four-wheel drive are other examples which are realized by embedded systems. The communication between these systems is not necessarily realized through a wireless connection, but through hard wired bus connections. The messages are generated by sensors or other controllers and are exchanged through a wide variety of different bus-systems, such as CAN [82], Flexray, and MOST [48]. To enable the different controllers on different bus-systems to communicate with each others, gateways are used to transfer messages to different bus-systems. These gateways manage the protocol conversions and also perform some special tasks, such as filtering messages, interpreting messages as well as performing routing and transmission of network messages on the different in-vehicle serial bus systems. These gateways are realized through microcontrollers, such as the 32-bit ARM7 microcontroller with IVN gateway functionality from Philips [122] and have a relatively high computation power (around 60 Mhz). These gateways also provide a programming interface through a serial port and can, thus, be re-programmed very easily [43]. However, no software protection methods are in place, and if an attacker is in possession of a controlling device, he can easily re-program the gateways of a vehicle. To make matters worse, no mechanisms are in place to protect messages while they are sent over the different bus-systems [113]. An attacker can, hence, eavesdrop on all traffic, inject packets, or replay old packets.

In the following, the gateways are also referred to as super node, since they perform similar operations as the super node in the scenario of WSN. In contrast to WSNs, super nodes (i.e., gateways) used in the automotive realm are very vulnerable to attacks, due to the severe consequences a faulty gateway can inflict. An attacker could, for example, re-program one gateway which would result in serious threats. Simply imagine wrong brake signals or wrong steering commands inflicted by a faulty gateway.

In contrast to the WSN scenario, the multihop communication is sparingly used in embedded systems. That is mainly caused by the fact that embedded networks seldomly cover wide areas. Thus, direct hard-wired connections or bus systems are often available. However, we will assume that multihop communication can also be applied in embedded systems, because the multihop communication provides a higher flexibility and more general solutions. Attestation protocols that enable secure integrity validation in multihop communication also enable secure integrity validation in a one-hop communication, but not vice versa.

We will mainly explain our proposed security mechanisms and the attestation protocols using the WSN scenario. However, all our developed security mechanisms can also be used in this scenario.

6.3 Assumptions

We assume that CNs are very limited in their storage and have low computational, communication, and/or energy resources. However, they have enough space to store a few bytes of keying information and are able to perform some basic operations, such as computing hash functions, symmetric encryption, etc. However, they are not able

to perform public key cryptography. The exact characteristic of such a node depends on the scenario. A node in the WSN scenario, for example, might be comparable to the Berkeley Mica Motes [41] and is very limited in its energy resources. In contrast, a node in the embedded systems scenario might not be that limited in its energy resources, since it does not necessarily depend on an own autonomic power supply.

SNs are assumed to possess much more computing power, memory capacity, and energy resources, e.g., comparable to the resources of the Stargate platform [42] or to the Philips gateways [122]. The TPM, integrated in the SNs, is used to protect keys and other security related data. We do not require any modification of the TPM, such as adding support for symmetric encryption with external data. Since present TPMs only support internal symmetric encryption, some data must be stored temporarily in the Random Access Memory (RAM) of a SN for further processing. For example, if the SNs are Linux-based systems (such as the stargate platform), all necessary driver modules and applications which allow access to the RAM should be disabled. Therefore, a static kernel has to be configured which prohibits the dynamic reloading of modules and all other possibilities to make a memory dump (e.g., file systems such as `/proc/kcore` has to be disabled). As soon as future TPMs support symmetric encryption with external data, the assumption that an adversary cannot access the RAM can be removed.

To subvert a SN, an adversary must re-program and reboot the node to either modify the system so that access to the RAM is possible or to access the security related data directly. After a reboot with a modified system, the platform configuration is changed and the access to sealed data is no longer possible. Thus, this data is neither accessible directly to the adversary nor loaded into the RAM. To achieve the binding of cryptographic keys to a specific platform configuration, which subsequently prevents rebooting in a compromised system configuration, we assume that we have a reduced measurement architecture, such as IBM's IMA [142], that extends the trust chain specified by the TCG up to the firmware and, therefore, includes integrity measurement of the kernel and operating system of the SN.

We assume an adversary who tries to compromise a SN to access stored information, e.g., keying material, and misuse the node to perform insider attacks, e.g., injecting false reports to cause false alarms. Therefore, the adversary can try to read out data or re-program the node to behave according to the purposes of the adversary. Furthermore, we consider an adversary based Dolev-Yao intruder model [50], who can eavesdrop on all traffic, inject packets, or replay old packets.

Since SNs are a valuable target for an adversary, it is reasonable to equip them with a TPM in scenarios where a high level of security is desired. CNs and a base-station or some sort of central processing unit should be able to verify whether a SN is still trusted, even if it is multiple hops away. We also refer to this base-station as *sink*. Since CNs are very limited in their resources, attestation protocols must be very lightweight, i.e., requiring only few, small messages and cheap operations (such as symmetric encryption). Sensed data is sent via multihop communication to the *sink*. The sink is assumed not to be constrained in its resources and cannot be compromised. It possesses all keying material shared with the sensor nodes.

Notation SNs are denoted as $SN_i, i = 1, \dots, a$ and the CNs are denoted as $CN_j, j = 1, \dots, b$, where $b \gg a$.

Applying a cryptographic *hash function* h on data m is denoted with $h(m)$. A one-way *hash chain* [98] stored on SN_i is denoted with $C^{SN_i} = c_0^{SN_i}, \dots, c_n^{SN_i}$. The hash chain is a sequence of hash values of some fixed length l generated by a hash function $h : \{0, 1\}^g \rightarrow \{0, 1\}^l$ where $g \gg l$ by applying the hash function h successively on a seed value $c_0^{SN_i}$ so that $c_{v+1}^{SN_i} = h(c_v^{SN_i})$, with $v = 0, 1, \dots, n - 1$.

A specific state of a SN_i is referred to as *platform configuration* $P_{SN_i} := (\text{PCR}[0], \dots, \text{PCR}[p])$ and is stored in the appropriate PCRs of the TPM. Data m can be cryptographically bound to a certain platform configuration P_{SN_i} by using the TPM.Seal command. Using the TPM.Unseal command, the TPM releases, i.e., decrypts m only if the platform configuration has not been modified. This concept allows an implicit attestation to be performed without a direct validation of the PCRs by a CN. Since we are abstracting the TPM.Seal and TPM.Unseal commands, we denote our commands with Seal and Unseal. Given a non-migratable asymmetric key pair (e_{SN_i}, d_{SN_i}) , we denote the *sealing* of data m for the platform configuration P_{SN_i} with $\{m\}_{P_{SN_i}}^{e_{SN_i}} = \text{Seal}(P_{SN_i}, e_{SN_i}, m)$. To *unseal* data m it is necessary that the current platform configuration P'_{SN_i} is equal to P_{SN_i} : $m = \text{Unseal}(P'_{SN_i} = P_{SN_i}, d_{SN_i}, \{m\}_{P_{SN_i}}^{e_{SN_i}})$.

6.4 Attestation Protocols

In this section, we describe our two proposed protocols which enable a CN to verify the platform configuration of a SN. These protocols represent some basic primitives which can be used in conjunction or in more complex protocols. To verify the trust level of received data from CNs, a SN has to perform additional mechanisms like redundancy checks or voting schemes.

We have adapted the sealing technique provided by the TPM to implement the implicit attestation (see Section 3.2). The characteristics of an implicit attestation do not require transferring the platform configuration (PCR values and SML). As a consequence, it is not required to satisfy Requirement 3 of a secure attestation channel which also does only apply for the explicit attestation. Our proposed attestation protocols consist of an initialization phase and an attestation phase. In the initialization phase the platform configuration of a SN is trusted. All data needed to perform a successful attestation is sealed in this phase to this platform configuration. Access to this sealed data is only possible if the SN is in the initial specified platform configuration. Compromising a SN results in a different platform configuration where access to this data is not possible. Thus, a successful attestation is no longer possible.

The first proposed protocol enables a broadcast attestation, where a SN broadcasts its platform configuration to its CNs in periodic intervals. This enables CNs to verify the platform configuration of the SN simultaneously. The second protocol enables a single CN (or the sink), to either individually verify the platform configuration of a SN using a challenge response protocol or to send data to a SN and receive a confirmation that the data has been received correctly and that the SN is trusted.

6.4.1 Periodic Broadcast Attestation Protocol (PBAP)

In some scenarios, many CNs perform measurements in parallel and in regular intervals. For example, a couple of CNs monitor the temperature in a specific region of the WSN. Another possible scenario could be that a couple of CNs realized as pressure sensor, monitor the tire pressure in a vehicle. The measurement is performed every 10 minutes to see the change over time. Therefore, the CNs report their measurement nearly in parallel and in specific time intervals to their SN. As a result, it might be desirable that all CNs are able to simultaneously verify if their SN is still trusted using a lightweight mechanism.

The PBAP adapts the idea of μ TESLA [119] to use one-way hash chains for authentication and extends it to enable attestation in hybrid WSNs. The sealing function of the TPM is used to bind a one-way hash chain to the platform configuration of a SN. A SN releases the values of the hash chain in periodic intervals, which can be verified by its CNs. This approach satisfies Requirement 1 of a secure attestation channel since this concept enables the CNs to verify the freshness of integrity information. The PBAP also integrates an authentication process to validate the authenticity of integrity information, which is achieved by exchanging cryptographic secrets between SN and the CNs in the secure initialization phase. However, Requirement 3 is not satisfied since no secure attestation channel is established which ensures that the further communication is bound to the currently attested SN.

The protocol is divided into two phases. In the *initialization* phase the SNs and the CNs are pre-configured before deployment. In the *attestation* phase SNs periodically broadcasts an attestation message. This phase normally lasts for the whole lifetime of the SNs.

Initialization Before SN_i is deployed, it is pre-configured with a non-migratable public key pair (e_{SN_i}, d_{SN_i}) and a SN_i specific hash chain C^{SN_i} . The seed value $c_0^{SN_i}$ of the hash chain is generated on SN_i using the TPM's physical random number generator and used by the CPU to perform the additional computations. SN_i is assumed to possess only one valid platform configuration, denoted as P_{SN_i} . After SN_i is powered up, a measurement about each component (BIOS, bootloader, operating system, applications) is performed, and the related values are stored in the corresponding PCR registers. Each value of the hash chain C^{SN_i} is sealed to this platform configuration P_{SN_i} :

$$\begin{aligned}
 \{c_0^{SN_i}\}_{P_{SN_i}}^{e_{SN_i}} &= \text{Seal}(P_{SN_i}, e_{SN_i}, c_0^{SN_i}) \\
 \{c_1^{SN_i}\}_{P_{SN_i}}^{e_{SN_i}} &= \text{Seal}(P_{SN_i}, e_{SN_i}, c_1^{SN_i}) \\
 &\vdots \\
 &\vdots \\
 &\vdots \\
 \{c_n^{SN_i}\}_{P_{SN_i}}^{e_{SN_i}} &= \text{Seal}(P_{SN_i}, e_{SN_i}, c_n^{SN_i})
 \end{aligned}$$

Each CN_j which interacts with SN_i is pre-configured with the last value $c_n^{SN_i}$ of the hash chain C^{SN_i} . Since the number of SNs is very small compared to the number of CNs, the CNs could be preprogrammed with the values of all SNs. After deployment, the CNs can only keep the values for its SN and another certain number of SNs in their vicinity to save memory.

Attestation Protocol 6.4.2 shows how SN_i periodically broadcasts its integrity. SN_i and the associated CNs (denoted as CN_*) are loosely time synchronized. The time is divided into intervals I_λ , $\lambda = 1, \dots, n$. At the beginning of each interval, SN_i sends a broadcast attestation message to the CNs. The attestation messages consist of the values of the hash chain released in reversed order of the generation and the identifier I_λ of the current interval. If the platform configuration of SN_i has not been modified, it can unseal the values of the hash chain C^{SN_i} . We will shortly explain the first attestation starting with the first interval I_1 .

In the first interval I_1 , SN_i unseals the hash value $c_{n-1}^{SN_i}$ and transmits it together with the interval identifier. In the second interval $c_{n-2}^{SN_i}$ is unsealed and transmitted and so on. CN_* check if the interval I_1 stated within the message matches their local interval counter I'_1 within a certain error range. If they match, CN_* verify whether $h(c_{n-1}^{SN_i}) = c_n^{SN_i}$. If the equation holds, SN_i is considered trusted and the value $c_n^{SN_i}$ is overwritten with the value $c_{n-1}^{SN_i}$. In the next interval SN_i releases $c_{n-2}^{SN_i}$ and so on, which are similarly checked. The protocol is shown in Protocol 6.4.2 and repeated from $\lambda = 1$ to n . n must be fixed before the network is deployed and large enough for performing enough attestations. In effect, the value of n describes how many PBAP can be performed.

Protocol 6.4.2: Periodic Broadcast Attestation Protocol

SUMMARY: The SN periodically broadcasts its integrity
 RESULT: Integrity reporting

1. *Notation.*

I_λ denotes a time interval

SN_i is a super node

CN is a cluster node

P_{SN_i} denotes the current platform configuration

2. *System Setup.*

SN_i and the associated CNs (denoted as CN_*) are loosely time synchronized. The time is divided into intervals I_λ , $\lambda = 1, \dots, n$.

3. *Protocol steps.*

Interval	Node(s)	Message or Action	
I_λ	SN_i	$\text{Unseal}(P_{SN_i}, d_{SN_i}, \{c_{n-\lambda}^{SN_i}\}_{P_{SN_i}}) = c_{n-\lambda}^{SN_i}$	(1)
I_λ	$SN_i \rightarrow CN_*$	$c_{n-\lambda}^{SN_i}, I_\lambda$	(2)
I_λ	CN_*	$I_\lambda \stackrel{?}{=} I'_\lambda$	(3)
I_λ	CN_*	if $h(c_{n-\lambda}^{SN_i}) \stackrel{?}{=} c_{lasthash}^{SN_i}$, SN_i 's state is valid	(4)
I_λ	CN_*	overwrite $c_{lasthash}^{SN_i}$ with $c_{n-\lambda}^{SN_i}$	(5)
...	

4. *Protocol actions.*

- (a) In the first step, the super node SN_i unseals the hash value $c_{n-1}^{SN_i}$.
- (b) The hash value $c_{n-1}^{SN_i}$ is transmitted to all CN_* .
- (c) CN_* check if the interval I_λ stated within the message matches their local interval counter I'_λ within a certain error range.
- (d) If the intervals match, CN_* verify whether $h(c_{n-\lambda}^{SN_i}) = c_{lasthash}^{SN_i}$. If the equation holds, the super node SN_i is considered trusted.
- (e) The value $c_{lasthash}^{SN_i}$ is overwritten with the value $c_{n-\lambda}^{SN_i}$.

If the communication between the CNs and the SN is implemented through wireless devices, a CN could miss some messages, e.g., due to unreliable communication. Thus, CNs should not immediately declare a SN as being untrusted but wait for a certain threshold of time. If a CN receives messages again, it can resynchronize by applying the hash function multiple times. Assume a CN misses the released values $c_{n-1}^{SN_i}$ and $c_{n-2}^{SN_i}$ in I_1 and I_2 . Receiving the value $c_{n-3}^{SN_i}$ in I_3 , the node can simply resynchronize and verify the platform configuration of SN_i by applying the hash function three times on $c_{n-3}^{SN_i}$ and verify whether $h(h(h(c_{n-3}^{SN_i}))) = c_n^{SN_i}$. But if the node does not receive any attestation messages after a certain number of intervals and/or the individual attestation fails, the CN should react by declaring SN_i as not trusted. CN could then, for instance, switch to a new SN.

6.4.3 Individual Attestation Protocol (IAP)

In contrast to the PBAP, the IAP can be used to individually verify whether the platform configuration of a SN is trusted. A CN needs only to perform symmetric operations and two short messages need to be exchanged. The messages are very small, because no long public key primitives, e.g., keys, signatures need to be transmitted. Since transmitting messages is the most cost intensive factor in WSNs [190], this is of particular

interest, especially if the sink wants to verify the platform configuration of a SN. In this case, messages are transferred along several hops.

The protocol we propose is again divided in *initialization* phase and *attestation* phase. The initialization phase is performed only once after deployment of the sensor nodes while the attestation phase can be performed every time a CN (or the sink) wants to verify the platform configuration of a SN. In the embedded world, this initialization phase could for example be performed directly before a vehicle leaves the factory site.

Initialization Each CN_j establishes a shared, symmetric key K_{CN_j,SN_i} with its SN_i . For this purpose, existing (non TPM-based) techniques, e.g., [191], might be used. However, we recommend using a key establishment protocol that uses the functionality of a TPM, e.g., [66]. This approach has the advantage that key generation within a TPM is inherently more secure than key generation on off-the-shelf embedded platforms. As in [191], we assume that this short period of time to establish pairwise keys is secure and nodes cannot be compromised. The keys K_{CN_j,SN_i} are sealed on SN_i to its valid platform configuration P_{SN_i} . Thus, SN_i can only access these keys if it is in its valid state.

To enable the sink to perform the attestation with SN_i , a shared symmetric key K_{Sink,SN_i} is preconfigured on SN_i before deployment and sealed likewise.

Attestation Protocol 6.4.4 shows how CN_j can verify the platform configuration of SN_i . First, CN_j sends a challenge to SN_i . The challenge consists of an encrypted block containing a *nonce* and the identifier ID_{CN_j} of CN_j , and an additionally ID_{CN_j} in plaintext. K_{CN_j,SN_i} is used for encryption. After receiving the challenge, SN_i unseals K_{CN_j,SN_i} related to ID_{CN_j} . This is only possible if the platform configuration P_{SN_i} is valid. Using this key, SN_i decrypts the encrypted block and verifies if the decrypted identifier is equal to the identifier received in plaintext. If they match, SN_i knows that this message originates from CN_j , encrypts the *nonce'* using K_{CN_j,SN_i} , and sends it back.¹ Otherwise, SN_i aborts. SN_i then deletes K_{CN_j,SN_i} from the RAM. CN_j decrypts the received response message and checks if the decrypted *nonce''* matches the *nonce* it has sent in the first step. If they match, SN_i is declared trusted and CN_j can send data to SN_i . This data is encrypted using K_{CN_j,SN_i} . The attestation of SN_i by the sink is performed likewise, using the key K_{Sink,SN_i} . Protocol 6.4.4 uses a challenge-response authentication involving the TPM to verify freshness of integrity information and thus satisfies Requirement 1. In addition, the ID of the CN is also injected to prevent replay attacks. Requirement 3 is also satisfied by binding further transaction data sent over the channel to a specific cryptographic key shared between the SN_i and CN_j . In contrast to the PBAP, this approach ensures authenticity of the attestation channel and, thus, establishes a secure attestation channel.

¹However, the trust level of CN_j cannot be assumed, because the node could be potentially compromised and the key is not protected by a TPM.

Protocol 6.4.4: Individual Attestation Protocol

SUMMARY: A CN verifies the integrity of a SN

RESULT: Integrity validation

1. *Notation.*

SN_i is a super node

CN is a cluster node

P_{SN_i} denotes the actual platform configuration

K_{CN_j, SN_i} denotes the shared key between CN_j and SN_i

2. *Protocol steps.*

$$CN_j \rightarrow SN_i : ID_{CN_j}, \{nonce, ID_{CN_j}\}_{K_{CN_j, SN_i}} \quad (1)$$

$$SN_i \quad \text{Unseal}(P_{SN_i}, d_{SN_i}, \{K_{CN_j, SN_i}\}_{P_{SN_i}}^{e_{SN_i}}) = K_{CN_j, SN_i} \quad (2)$$

$$SN_i \quad D(\{nonce, ID_{CN_j}\}_{K_{CN_j, SN_i}}, K_{CN_j, SN_i}) = (nonce', ID'_{CN_j}) \quad (3)$$

$$SN_i \quad \text{check } ID'_{CN_j} \stackrel{?}{=} ID_{CN_j} \quad (4)$$

$$SN_i \quad E(\{nonce', ID_{SN_i}\}, K_{CN_j, SN_i}) = \{nonce', ID_{SN_i}\}_{K_{CN_j, SN_i}} \quad (5)$$

$$SN_i \rightarrow CN_j : ID_{SN_i}, \{nonce', ID_{SN_i}\}_{K_{CN_j, SN_i}} \quad (6)$$

$$SN_i \quad \text{delete } K_{CN_j, SN_i} \text{ from RAM} \quad (7)$$

$$CN_j \quad D(\{nonce', ID_{SN_i}\}_{K_{CN_j, SN_i}}, K_{CN_j, SN_i}) = (nonce'', ID'_{SN_i}) \quad (8)$$

$$CN_j \quad \text{if } nonce'' \stackrel{?}{=} nonce, \text{ state of } SN_i \text{ is valid} \quad (9)$$

3. *Protocol actions.*

- (a) In the first step, CN_j transfers an encrypted nonce together with its ID using the shared key to SN_i .
 - (b) SN_i performs the `TPM_Unseal` command and unseals the shared key.
 - (c) SN_i decrypts the sent message and verifies in (4) whether the ID of the decrypted message matches the ID sent in plaintext.
 - (d) SN_i encrypts the decrypted nonce and the ID of CN_j with the unsealed key. The obtained message is transferred to CN_j in (6).
 - (e) SN_i deletes the unsealed key from its RAM.
 - (f) CN_j decrypts the received message and verifies (9) if the received nonce matches the nonce sent in (1).
-

It might also be preferable to directly transmit data in an attestation challenge rather than waiting until an attestation response has been received and a SN is declared

trusted. This might be preferable in scenarios where an immediate receipt of data is important or where CNs send data very infrequently to a SN. Therefore, the protocol is modified in steps (1) and (3). Protocol 6.4.5 shows the resulting protocol and also satisfies Requirement 1 and Requirement 3 of a secure attestation channel (see Chapter 3. In step (1) CN_j sends the data to SN_i within the encrypted block. SN_i can only decrypt this message in step (3) if its platform configuration is valid and access the data. All other steps remain the same as shown in Protocol 6.4.4. Thus, if CN_j receives the message in step (6) and the checks in steps (7) and (8) succeed, CN_j can be assured that SN_i has successfully received the data and is still trusted.

Protocol 6.4.5: Modified Individual Attestation Protocol

SUMMARY: A CN delivers data that is bound to a specific state to a SN

RESULT: Data transfer with data confirmation and integrity reporting

1. *Notation.*

SN_i is a super node

CN is a cluster node

P_{SN_i} denotes the actual platform configuration

K_{CN_j, SN_i} denotes the shared key between CN_j and SN_i

2. *Protocol steps.*

$$CN_j \rightarrow SN_i : ID_{CN_j}, \{nonce, ID_{CN_j}, data\}_{K_{CN_j, SN_i}} \quad (1)$$

$$SN_i \quad \text{Unseal}(P_{SN_i}, d_{SN_i}, \{K_{CN_j, SN_i}\}_{P_{SN_i}}^{e_{SN_i}}) = K_{CN_j, SN_i} \quad (2)$$

$$SN_i \quad D(\{nonce, ID_{CN_j}, data\}_{K_{CN_j, SN_i}}, K_{CN_j, SN_i}) = (nonce', ID'_{CN_j}, data) \quad (3)$$

$$SN_i \quad \text{check } ID'_{CN_j} \stackrel{?}{=} ID_{CN_j} \quad (4)$$

$$SN_i \quad E(\{nonce', ID_{SN_i}\}, K_{CN_j, SN_i}) = \{nonce', ID_{SN_i}\}_{K_{CN_j, SN_i}} \quad (5)$$

$$SN_i \rightarrow CN_j : ID_{SN_i}, \{nonce', ID_{SN_i}\}_{K_{CN_j, SN_i}} \quad (6)$$

$$SN_i \quad \text{delete } K_{CN_j, SN_i} \text{ from RAM} \quad (7)$$

$$CN_j \quad D(\{nonce', ID_{SN_i}\}_{K_{CN_j, SN_i}}, K_{CN_j, SN_i}) = (nonce'', ID'_{SN_i}) \quad (8)$$

$$CN_j \quad \text{if } nonce'' \stackrel{?}{=} nonce, \text{ state of } SN_i \text{ is valid} \quad (9)$$

3. *Protocol actions.*

(a) In the first step, CN_j transfers an encrypted nonce together with its ID and the sensed data using the shared key to SN_i .

(b) SN_i performs the `TPM_Unseal` command and unseals the shared key.

- (c) SN_i decrypts the sent message and verifies in (4) whether the ID of the decrypted message matches the ID sent in plaintext.
 - (d) SN_i encrypts the decrypted nonce and the ID of CN_j with the unsealed key. The obtained message is transferred to CN_j in (6).
 - (e) SN_i deletes the unsealed key from its RAM.
 - (f) CN_j decrypts the received message and verifies in step (9) whether the received nonce matches the nonce sent in (1).
-

6.5 Analysis

In this section, we first discuss the security of the two proposed attestation protocols. Then, we evaluate their performance.

6.5.1 Security Discussion

The goal of both protocols is that CNs can prove the trust level of SNs. If an adversary compromises a SN, he cannot successfully deceive the CNs or the sink to perform insider attacks. We distinguish between two types of possible attacks:

- (1) attacks against a SN directly and
- (2) en-route attacks if the communication involves multiple hops.

Due to the unreliable multihop communication and the assumption that our communication channel is insecure, we can only prove the trust level of SNs. But we cannot prove whether a node is not trusted since either communication errors can result in modified attestation messages or malicious en-route nodes can modify forwarded messages to defame a SN. This is different to most approaches of the software-based attestation [150, 146] where a monitored and secured communication channel is assumed. If our communication channel is also assumed to be secure, our protocols can be used to validate whether a specific SN is not trusted. However, we believe that this assumption is too restrictive and not very appropriate. Thus, our proposed attestation protocols do not give a statement about the trust level of a used route. In addition, an invalid attestation could be caused either by a compromised SN or by a compromised en-route CN.

Security of the PBAP

To compromise a SN and forge a trusted platform configuration, an adversary needs access to the hash chain. Therefore, he has to either perform the unseal command under a compromised platform configuration, or try to access the key used to seal the hash chain with physical attacks. As described in Section 2.1, the TPM is basically a

smart card and offers high security mechanisms for preventing unauthorized extraction of protected keys. This makes it extremely difficult for an adversary to retrieve the necessary keys to decrypt the sealed hash chain. Additionally, access to the sealed hash chain is only possible if the platform configuration has not been modified. This prevents the unauthorized extraction of the values of the hash chain in a compromised system environment. Even if an adversary could access the RAM of a sensor node, he cannot retrieve other hash values, because for each attestation only the current hash value is unsealed and loaded into the RAM.

However, our approach cannot handle runtime attacks caused by buffer overflows, since we report the platform configuration measured in the initialization phase, i.e., when the software is first executed. Such attacks would result in a (malicious) modified system configuration, but the platform configuration stored in the PCRs is still the valid configuration.

If the attestation is performed between nodes which are multiple hops away, an adversary might also try to perform a man-in-the-middle attack by compromising an en-route CN. The adversary can try to spoof, alter or replay attestation messages, or perform a selective forwarding attack [90]. Spoofing is not possible, because PBAP is not an authentication protocol. It gives an assertion about the trust level of the specific SN and not which node has relayed the message. Altering attestation messages is possible and results in an unsuccessful attestation. To cope with that, a CN should possess an additional mechanism which enables the CN to reach its SN using a different communication path or change to a different SN. The CN can then use an alternative path and perform the IAP with the SN to make a clear statement, whether the route, or the node has been compromised. If the SN is compromised, a CN could, for example, switch to another SN where the communication paths and the new SN may not have been compromised. Replay attacks or an attack where an adversary first blocks the forwarding of legitimate hash values to collect them, then compromises a SN and finally releases these hash values are not possible, because hash values are only valid for a specific interval, which is validated by each CN. Since the PBAP is performed in plaintext an adversary can distinguish between attestation and data messages and therefore perform a selective forwarding attack by forwarding attestation messages, but blocking data messages. Such attacks are a general problem in WSNs and show that the PBAP is not resistant against all attacks in a multihop scenario with malicious en-route CNs.

Security of the IAP

The security of the IAP relies on the sealing of the symmetric keys to the valid platform configuration analogue to the sealing of the hash chain described above. Thus, an adversary compromising a SN cannot access the necessary keys to perform a successful attestation.

If the attestation messages are forwarded along multiple hops, an adversary can try to perform a man-in-the-middle attack. Since the IAP includes an authentication protocol, spoofing is not possible. A SN detects the modification of the first attestation

message (see Protocol 6.4.4) by an en-route adversary, since the included identifier does not match the identifier sent in plaintext. If the adversary alters the response sent to a CN, the latter cannot distinguish if either the attestation has failed or if the message has been altered by the adversary. Replay attacks are not possible, because a new nonce is used in each message. Since attestation messages and data messages have the same form (identifier plus encrypted data block), an adversary cannot distinguish between them to perform a sophisticated selective forwarding attack. If the modified IAP is used, where data is sent in the first step, an adversary might be able to distinguish between this message and the response message (step 6) because of the different lengths of the messages. To cope with that, the message sent in step 6 could be padded to the same length.

Thus, if an attestation fails, a CN should first try to perform a new attestation of the same SN using another communication path, if possible. If this is not possible or the attestation fails again, either the SN or a node on the communication path is compromised. The CN should then select a new SN, since messages sent to the old one might be susceptible to attacks.

Furthermore, in contrast to scenarios where SNs are not equipped with a TPM, a single compromise of a SN does not result in the compromise of all shared keys stored on this node. Even using the TPM in only a few nodes results in a higher resiliency to node compromise.

6.5.2 Performance Analysis

Efficiency is crucial for security protocols for WSNs as well as embedded systems because of the limited resources of nodes. Protocols should not introduce a high storage overhead and should not significantly increase energy consumption. Since we assume that SNs possess sufficient resources, we perform our analysis only for the CNs. We restrict us to only perform an analysis for the WSN scenario, since these devices typically possess lower computation power and lesser power supply. For this purpose, we first analyze the additional storage requirements. Next, we estimate the additional energy consumption by evaluating the computational and communication overhead.

Storage Requirements

For the PBAP, a CN must store one hash value and the identifier for the corresponding SN. Depending on the network configuration, it might also store hash values (and identifiers) for other SNs in its vicinity. Let L_N , and L_H denote the length of a *node identifier* and a *hash value* respectively. Let the number of SNs for which a CN stores values be v . Thus, the storage requirements SR_{PBAP} for a CN are:

$$SR_{PBAP} = v * (L_N + L_H). \quad (6.1)$$

For example, suppose a CN stores values for 5 SNs. The length of each hash value is 64 bits and the length of a node identifier is 10 bits. This results in a total of 46.25 bytes.

For the IAP, a CN must store one symmetric key for each SN with which it wants to perform an attestation. Let this number be denoted by w and the length of a key denoted by L_K . Thus, the storage requirements SR_{IAP} for a CN are:

$$SR_{IAP} = w * L_K. \quad (6.2)$$

For example, suppose a CN stores keys for 5 SNs. The length of each key is 64 bits. This results in a total of 40 bytes.

The Berkeley Mica2 Mote [41] offers 4KB of SRAM. As a result, a CN of type Berkeley Mica2 Mote can store 512 keys, giving him the ability of performing the IAP with 512 different SNs. In case of the PBAP, he can store approx. 443 hash values, thus, a CN is able to attest 443 different SNs using the PBAP. Therefore, the storage requirements are suitable for current sensor nodes, even if both protocols are used in conjunction.

Energy Consumption

The PBAP requires a CN to receive one attestation message and to perform one hash computation at each time interval. An attestation message consists of a hash value and an identifier of the interval, e.g., a counter. Although computing hash values only marginally increases energy consumption [119], we consider the computational overhead, since a hash computation is performed in each time interval.

We use $e_1 = e_{1s} + e_{1r}$ to denote the energy consumed in sending and receiving one byte, and e_2 to denote the energy for one hash computation. In addition to the notation used above, let L_T denote the length needed for the interval identifier. The total number of intervals in the whole lifetime of the network is denoted with t . This results in an additional energy consumption:

$$E_{PBAP} = t * ((L_T + L_H) * e_{1r} + e_2). \quad (6.3)$$

For example, suppose the lifetime of the network is one year and broadcast messages are sent every 10 minutes. Therefore, a 16 bit counter is sufficient for numbering each interval. We use the results presented in [190] to quantify $e_{1s} = 16.25 \mu J$ for sending, $e_{1r} = 12.5 \mu J$ for receiving, and $e_1 = 28.75 \mu J$ for sending and receiving one byte using Berkeley Mica2 Motes. The energy consumed for performing one hash computation using RC5 [127] block cipher is $e_2 = 15 \mu J$. This results in a total energy consumption of 7358.4 mJ. The Mica2 Motes are powered with two 1.5 V AA batteries in series connection. We assume a total capacity of 2750 mAh using standard AA batteries which results in 29700 J. Thus, the ratio of energy consumed in one year by the PBAP is about 0.025% of the total available energy which is negligibly small.

The IAP requires a CN to generate and send a challenge², and the verification of the response (see Protocol 6.4.4, steps 1, 6, 8 and 9). The challenge requires one nonce generation, one encryption and one transmission, while the response verification

²We do not consider the case where data is sent within the challenge, because we estimate only the additional overhead introduced by our protocol.

requires the receipt of one message, one decryption and one comparison of two values. As in [120], the nonce is generated using a Message Authentication Code (MAC) as pseudo-random number generator with a generator key $K_{CN_j}^{rand}$. The energy consumed using RC5 for MAC generation is $e_2 = 15 \mu J$. The encryption cost using RC5 are also $15 \mu J$. We neglect the energy cost for the comparison of two values since they are negligibly small. Thus, the additional energy consumption is:

$$E_{IAP} = 3 * e_2 + (e_{1s} + e_{1r}) * (2 * L_N + L_H). \quad (6.4)$$

Assuming the values from above, this results in a total energy consumption on a CN for one individual attestation of about $347 \mu J$ which is $1.17 * 10^{-6}\%$ of the total available energy. As a result, a CN can execute the IAP approx. 85.6 million times.

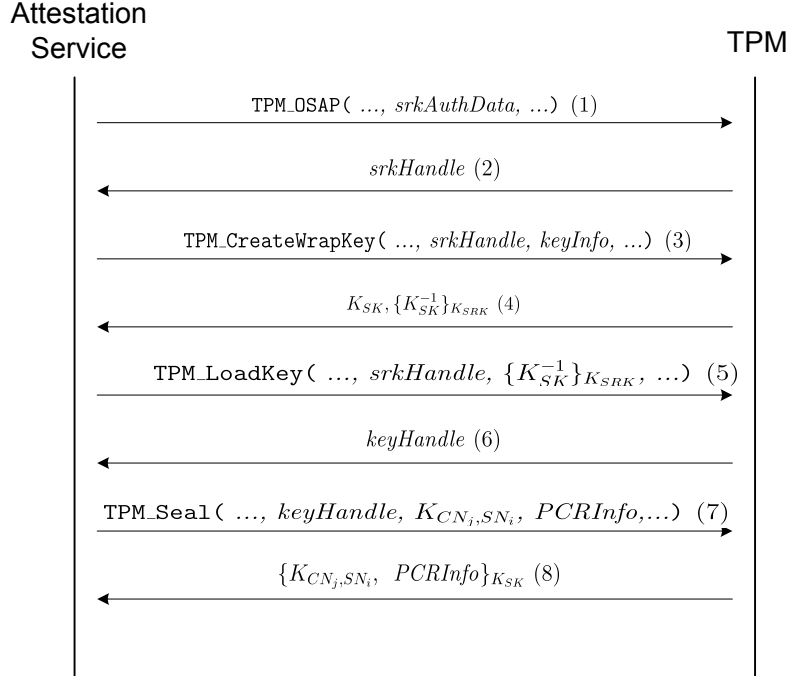


Figure 6.2: Communication between the TPM and an attestation service to realize the initialization phase of the IAP

6.6 Implementation

In order to run simulations about the performance and the behavior of an attestation-enhanced WSN in the presence of an adversary, the protocols proposed in this chapter have been implemented [51] using tpm4java framework³ and J-SIM⁴. For this purpose,

³<http://tpm4java.datenzzone.de>

⁴<http://www.j-sim.org/>

the J-SIM environment has been modified and enhanced to support three different types of additional nodes. The first node type is the source CN, which receives attestation messages or creates attestation challenges and forwards them in the direction of the SN. The second node type is a relaying node, that only receives a message and relays this message in the direction of the SN. The third node type is the SN. All SNs run an attestation service that is responsible for the communication to the TPM of the SN. To execute the attestation protocols, a network consisting of these three types of nodes is created using the J-Sim framework. Then, the attestation service running on the SNs execute the initialization phase in which the keys (K_{CN_j, SN_i} with $j \in 1, \dots, b$) or the hash chain ($C^{SN_i} = c_0^{SN_i}, \dots, c_n^{SN_i}$) are sealed to the initial (trusted) platform configuration of the SN. In the following, we restrict us to shortly discuss the implementation of the IAP initialization phase. All other protocols are implemented likewise and only require that the message is delivered to a specific node. Further details about the implementation of these protocols can be found in [51]. Figure 6.2 shows the involved TPM commands necessary for realizing the initialization phase of the individual attestation phase. The attestation service first establishes a TPM Object-Specific Authorization Protocol (OSAP) session to the TPM (1). This command requires authorization (owner-password and srk-password) and computes a cryptographic secret and a handle (*srkHandle*) to this session (2); both to protect the whole session traffic (Cf. [171], pp. 71). The returned handle is then used to issue the `TPM.CreateWrapKey` command (3). This command generates a TPM key which is used to encrypt all TPM sealed data. This generated key is first loaded (5) and then used to seal all shared symmetric keys between the SN and the CNs (7).

6.7 Summary

In this chapter, we showed how attestation techniques can be realized in systems with very low computation power. In this context, we presented two attestation protocols for a network organized in clusters. The network consists of resource constrained cluster nodes and super nodes with more resources equipped with a TPM-chip that acts as a trust anchor. The first protocol provides a broadcast attestation, i.e., allowing a super node to attest its system integrity to multiple cluster nodes simultaneously, while the second protocol is able to carry out a direct attestation between a single cluster node (or the sink) and one super node. Both protocols allow cluster nodes to verify whether the platform configuration of a super node is trusted or not, even if they are multiple hops away. We have also shown in this chapter, that both the overhead for storage and the energy consumption are negligible.

Part III

Attestation-supporting Security Architecture

Chapter 7

Security Architecture

In this chapter, we present the security architecture for trusted transactions in highly sensitive environments. This architecture enables leveraging attestation techniques for trust establishment in distributed systems, so as to guarantee that the system is free of malware and that its software has not been tampered with. Most parts of this chapter have been previously published in *An Approach to a Trustworthy System Architecture Using Virtualization* [156], *Towards Secure E-Commerce Based on Virtualization and Attestation Techniques* [158] and *An Architecture providing Virtualization-Based Protection Mechanisms against Insider Attacks* [161]. However, these publications did not include a detailed evaluation of the security architecture.

7.1 Introduction

In the first part of this thesis, we analyzed how attestation techniques can be used to ensure a trusted and malware free platform. For this purpose, we proposed a number of protocols that enable establishing secure attestation channels and can be used by a second entity for placing trust in a remote entity's system configuration. In this context, we showed how secure attestation channels can be established, how the scalability of attestation techniques can be improved, and how an efficient attestation can be realized and the platform validation process be simplified. However, using one of these proposed attestation protocols in an open-, non-constrained software system is impractical. This is due to the fact that currently available and wide-spread operating systems are very complex and not very reliable. They suffer from fundamental design flaws because of their monolithic design concept. Monolithic operating systems tend to integrate all services in the base kernel and, thus, overload the kernel with functionalities that run at the highest privilege level, causing the approach to break the principle of least privilege [143]. In addition, such systems do not provide strong isolation between execution environments. As a result, in order to use attestation techniques to ascertain that a specific target application is trusted, all executed software components of the operating system need to be fully trusted, even if they have no direct impact on the target application.

To make matters worse, it is necessary to measure all executed operating system components, resulting in hundreds of measurements, each requiring that a trusted reference value be provided. This shows that it is not feasible to use attestation techniques in currently available, non-constrained operating system environments.

In this chapter, we present a security architecture for non-resource constrained platforms that does not possess the drawbacks of current operating systems. To this end, we show how the security of operating system environments can be enhanced by establishing multiple isolated execution environments with different trust levels. These separate environments are created by several different virtual machines (VM) running on a single system. This approach keeps certain instances separate, giving us the ability to attest to one instance at a time rather than to the whole operating system environment. Like Terra [67], we use different types of virtual machines to realize isolated compartments: A *trusted VM* which is responsible for running highly sensitive code, an *open VM*, which runs arbitrary software components, and a *management VM*, which is responsible for spawning new VMs. Besides presenting the operating system environment, we will also show how attestation techniques can also be used to secure the operating system environment. In this context, we will show that combining virtualization and Trusted Computing eliminates intrinsic problems that each faces when used alone.

This chapter is organized as follows. In Section 7.2, we present the challenges that our security architecture must solve. Chapter 7.4 shows the basic design concepts of our approach and Section 7.5 presents the detailed security architecture and the security mechanisms of our approach. In Section 7.6 we show how one virtual machine can report its integrity to a remote entity in order to establish trust in the respective system configurations. Section 7.7 evaluates our architecture and we analyze whether the challenges presented in Section 7.2 are solved. Finally, we summarize this chapter in Section 7.10.

7.2 Security Challenges

In this section, we present the challenges our security architecture must solve. In this context, we first show where subversive programs such as malware and spyware typically attack when trying to compromise a system. Based on this foundation, we present our considered adversary model. Finally, we will show how attestation techniques can be used to detect whether a system is in an uncompromised state and which further restrictions need to be solved.

7.2.1 Malware Attack Points

Before we construct and present the system architecture, we will look at potential attacks that threaten current operating systems and their applications. In this context, we assume that a simple application that enables secure transactions between particular platforms is running on an arbitrary operating system, such as Linux or Windows. A particular user uses the application to enter and transfer confidential data, e.g., a password, to a communication partner. In the following, we denote such an application

as a secure transaction client. A transaction is, thus, the transmission of confidential information to another communication partner. Using this scenario, we will describe where the vulnerabilities are and present typical attack points. For this purpose, we adapt and extend the classification that we proposed in [161] and combine it with the classification proposed by Grawrock in [73]. The resulting classification consists of four different categories: Software Manipulating Attacks, Hardware Attacks, Input attacks and Output Attacks.

Software Manipulation Attacks

Software manipulation attacks try to access the memory that is allocated by the secure transaction client to gain access to the stored password. This is the typical attack vector of malware. These attacks mostly try to execute a malicious software component in ring 0 of the CPU. In this highest CPU privilege level, it is possible to access all running programs, including the secure transaction client. The operating system, as well as device drivers, typically run in this highest privilege ring. An attacker could, therefore, try to install a malicious device driver in order to gain access to the whole memory.

Hardware Attacks

Hardware attacks are attacks that exploit the characteristic of hardware components in order to get access to protected data. One example of such an attack is when a device with direct memory access (DMA) is used to extract confidential data. DMA allows hardware subsystems (especially peripherals) to directly access the system memory for reading or writing. DMA bypasses any protection managed by the CPU and allows access to the entire memory. Thus, an attacker can manipulate a device to perform a malicious DMA access to a memory region currently allocated by the secure transaction software. Note that this class of attacks does not include hardware attacks that cannot be conducted by malware, such as, mechanical or electrical probing attacks on the hardware components.

Manipulated Input

This class of attacks includes all attacks that occur on the software interface of information input. This comprises attacks that try to access secret data by capturing keystrokes using software mechanisms. This can be achieved by manipulating the existing keyboard device driver or by installing a key logger. This key logger need not necessarily be executed in ring 0 as shown in [29], but can also be executed in a lower CPU privilege level.

Manipulated Output

This class comprises attacks that occur on the software interface of information output. This includes attacks that are able to manipulate the output in attempt to trick the user into entering his password into a malicious program. Typical attacks of this type

are phishing attacks [44], screen scraping, changing the display driver, changing the X-Server or altering transaction data [99].

In addition, server spoofing falls into this category. An attacker could try to masquerade a server, e.g., by redirecting the network traffic using a DNS poisoning attack or by modifying the `/etc/hosts` file and, thus, force a user into entering or transferring confidential data to this server.

7.2.2 Attacker Model

We assume an adversary who tries to access sensitive transaction data entered by a particular user into the simple application for secure transactions. The adversary is able to perform all attacks discussed in Section 7.2.1 as well as the relay attack presented in Chapter 3 to circumvent potentially used attestation techniques. However, the adversary is not able to perform attacks that require physical access to the machine. We also assume that the attacker is not able to find collisions of the hash-function used by the TPM and that the hash-function is *2nd pre-image resistant* [131], i.e., it is infeasible for an attacker to find for a given x another x' with the characteristic that $f(x) = f(x')$, where f is the cryptographic hash-function.

7.3 Ensuring System Integrity using Attestation Techniques

To detect that a system is infected by malware, the TPM-based binary attestation could be used. To realize the TPM-based binary attestation, a measurement architecture [142] is required that measures every executed binary and adds the obtained measurements to the PCRs of a TPM. In addition, each executed software component's execution is recorded by the stored measurement log (SML). Therefore, the SML and the PCRs reflect the current state of a platform. An attestation interface is then used to establish a secure channel (Compare Section 3) and to transfer the measurements to a verifier using platform attestation. As a result, the verifier receives a list of all binaries that have been executed since the last reboot of the machine and is able to determine whether a platform is trusted. If the platform has been infected by malware, it is not trusted and the verifier will not let the user enter his confidential data¹. However, this approach has three problems:

Time Discrepancies The first problem is that the load-time measurement does not correctly reflect the runtime-behavior, since the received list of executed binaries gives no hint about what was executed after the remote attestation. This fact is similar to the problem of the time-of-use and the time-of-attestation discrepancy [152, 83], also often referred to as time-of-check and time-of-use (TOCTUE) problem [145]. This discrepancy addresses the problem that a software component may be correct at the

¹It should be noted that this is simplified, since a successful platform attestation could have also been simulated. We will discuss this in detail in Chapter 9 where we also present a solution for this restriction. However, for the sake of simplicity, we assume at this place, that this attack can be prevented using the described process.

time of attestation, but not at the time the code is used. In this context, we mainly encounter two types of problems:

1. A software component, such as a key logger or Trojan horse may be executed after the remote attestation. Although the malware is located on the client machine at the time of the attestation, it was not detected, because it was executed after the attestation took place.
2. A component's integrity may be corrupted through runtime attacks, such as buffer overflows, after the component has been measured and executed.

Incomplete Measurements The second problem is that only the executable binaries of applications are inspected. Shell scripts and configuration files are not included in the measurement. This is due to the fact that especially for arbitrary shell scripts or configuration files, reference values can hardly be provided. This leads to the problem that certain manipulations, e.g., manipulations of the configuration files, cannot be detected by this approach.

Inefficiency Finally, the approach is inefficient, since it requires that all measurements be known and fully trusted, even if they do not have a direct impact on the application whose trust level is to be evaluated [83]. This is caused by the fact, that all processes running on the machine can mutually influence each another [105, 84]. A process running in the privileged-mode of the operating system, such as a device driver, is able to inspect all memory regions that are allocated by another process.

To solve these challenges new approaches are needed that are able to reduce the complexity and allow one to measure the complete system configuration including scripts and configuration files.

7.4 Design Concept of the Security Architecture

Virtualization techniques have already been proposed in the literature [167] to increase safety, reliability and dependability. In this context, these techniques are adapted to provide isolated compartments that cannot be influenced by other isolated compartments running on the same system. Since virtualization is a very old concept, these techniques have also been significantly investigated in the literature (compare [69, 61, 1, 52, 183, 132]).

However, virtualization techniques can also be used to leverage attestation techniques for trust establishment. The use of this concepts enables the reduction of the complexity of the attestation process. Instead of attesting to the whole platform configuration, including all processes running on the machine, our approach only attests to processes that are required for trusted operations. Processes that are not responsible for establishing communication with a remote entity, and therefore not required for the transaction, i.e, the transmission of confidential data, itself, should neither be included in the integrity measurement nor be able to influence the transaction. The integrity

measurement and the integrity reporting should, therefore, only include sensitive processes that are vital for the whole transaction and should exclude other applications, such as e-mail, multimedia-applications, or office-applications.

Using virtualization techniques also enables a reduction in the amount of information needed to describe the trust-level of a platform configuration. As a consequence, a verifier is not able to discover the full platform configuration, including all running processes, but only a subset of the full platform configuration. This approach, therefore, satisfies potential emerging privacy-concerns.

To establish different execution environments and to only attest fragments of the whole platform configuration, we use concepts similar to those provided in Terra [67]. The main difference is that our integrity measuring and reporting facilities are based on techniques introduced by the TCG, thus, benefiting from a hardware-based trust anchor. In contrast to Terra, we also show how trust can be guaranteed using secure integrity protocols.

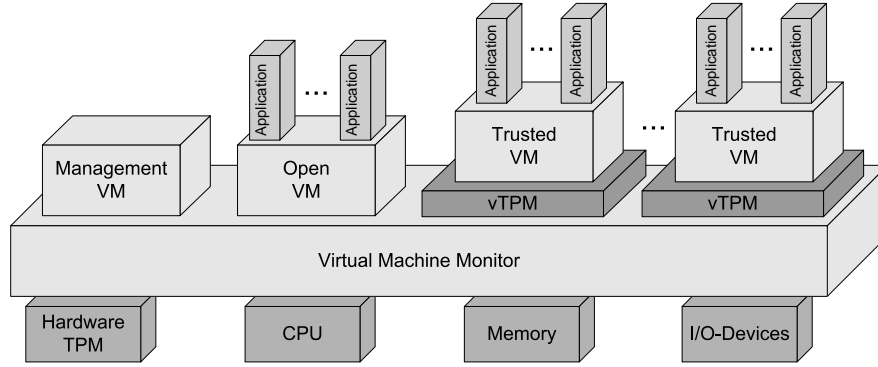


Figure 7.1: Basic design concept of the security architecture

Figure 7.8 depicts the design principles of our security architecture. It consists of three different types of virtual machines, a hypervisor for partitioning the underlying hardware, and a hardware-based trust anchor. The *trusted virtual machine monitor* (TVMM), or hypervisor, forms the foundation for our architecture by providing an abstraction layer for the underlying hardware. We will shortly describe these different components before we go in detail.

7.4.1 Overview of the Trusted VMM

The trusted VMM has privileged access to the hardware and is able to grant and revoke resources to and from the running VMs (e.g., CPU scheduling). Because of its privileged position, the VMM is able to inspect every single CPU cycle of each virtual machine, therefore, needs to be trusted. We assume that the VMM is trusted and that it reliably provides the properties of a VMM, i.e., strong isolation. Currently available virtualization solutions provide strong isolation. However, strong isolation, can still be circumvented with direct memory access (DMA) operations [62]. DMA

operations access the memory without intervention by the CPU and, therefore, bypass the hypervisor's protection mechanisms. In hypervisors with *secure sharing* [89], the requires I/O emulation is moved into the hypervisor layer, allowing them to prevent such attacks; however, such hypervisors suffer from a high performance overhead and a large Trusted Computing Base [140].

7.4.2 Overview of the Trust Anchor

In order to be able to report the system configuration of the virtual machines, a hardware-based trust anchor, i.e., the Trusted Platform Modules is introduced. However, the TPM was never designed to be used in virtual environments, and is, thus, not capable of being used in a system that is based on virtualization. This restriction results from the fact that the TPM holds state specific data. If two different VMs are accessing the same TPM, one VM could change the state of the TPM, which would also change the state of the other virtual machine's TPM. To overcome this restriction, we make use of the approach by Berger et al. [20] who propose to virtualize the hardware TPM, by equipping every VM with its own software TPM. This software TPM only uses the underlying hardware TPM for certain operations. Access to this virtual TPM is controlled by the VMM which provides an abstraction of the underlying hardware TPM through a virtualized TPM interface. This virtual TPM (vTPM) is mainly used by the *trusted virtual machine* (TVM), which, in turn, is used for trusted transactions with a remote entity.

7.4.3 Virtual Machines

The VMM establishes several different execution environments by using various types of virtual machines that are strongly isolated from one another. A virtual machine only represents the interface for the bare hardware constructed by the VMM [69], and each virtual machine needs its own software environment. Such software environments are referred to as virtual appliances [144] and consist of a fully pre-installed and pre-configured application and operating system environment. Our security architecture consists of three different types of virtual machines.

Overview of the Trusted VM The TVM handles private and sensitive data and runs a tiny OS with a minimal number of processes, thereby considerably reducing the possible number of security vulnerabilities. The software image is supplied by a trusted third party and consists of the TVM as well as the transaction client. Before startup, the software image is measured by a measuring process before using the hardware hashing engine of the TPM. These measurements are then made accessible inside the virtual machine by adding them into the PCRs of the vTPM. For attestation, the trusted VM transmits these measurements to a remote entity by accessing them through the vTPM instance.

Open Virtual Machine The *open virtual machine* is allowed to run arbitrary software components and provides the semantics of today's open machines. It runs applications with a lower trust grade, such as web-browsers or office applications. The advantage of using an open VM is twofold. First, using an open virtual machine gives us the ability to provide a software environment that can be used for any arbitrary application. This enables us to construct an environment where full-open commodity applications can be used without influencing the TVM. Second, security breaches by compromised applications are restricted to the Open VM. These security breaches are very likely, since the environment consists of multiple different applications with a wide spectrum of security vulnerabilities.

Management Virtual Machine The *management virtual machine* is responsible for starting, stopping and configuring the virtual machines. It provides security services to the trusted VM, such as VM image integrity measurement and secure storage. The management VM is closely connected to the virtual machine monitor, since it is a privileged VM and has direct access to the hardware TPM.

7.5 Security Architecture

Our proposed security architecture and the involved security services are shown in Figure 7.2. In this section, we will explain the components of our architecture and also its underlying security mechanisms in detail.

7.5.1 Trusted Virtual Machine Monitor and Management VM

The TVMM and the management VM form the foundation of our security architecture. The TVMM is a software layer that offers an interface to the hardware for the virtual machines currently running on the platform. In general, the only purpose of a VMM is to partition the underlying hardware and to provide an interface for the generic operation systems running in the individual virtual machines. The properties of a virtual machine monitor have been extensively studied in the literature (e.g., [24, 94, 32, 185, 180]), providing isolation, efficiency, compatibility and simplicity. In our security architecture, the TVMM is closely connected to the management VM so as to provide additionally security services to the virtual machines. These security services are: integrity measurement, secure storage and TPM support. The security architecture, thus, has the following characteristics:

Attestation The virtual machines are able to access the underlying Trusted Platform Module without modifying the state of the TPM; if the state of the TPM were modified, other virtual machines would be affected. An attestation service running inside the virtual machine is able to report the local system state to a remote party that authenticates the local system state. This, in turn, allows a remote entity to place trust in the system configuration of a communication partner. This property is achieved

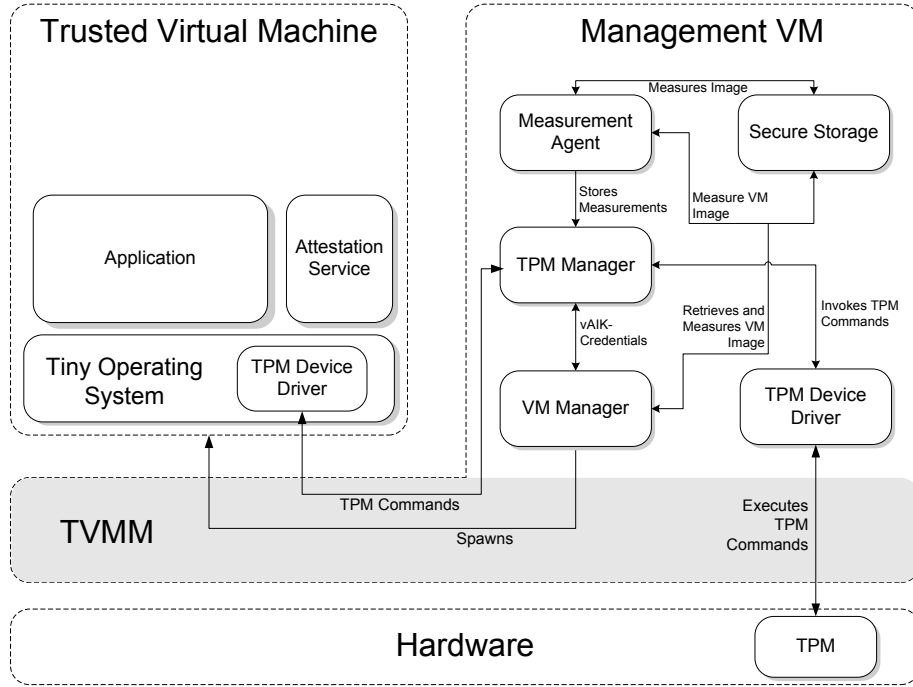


Figure 7.2: The proposed security architecture

by virtualizing the TPM and by providing a software TPM for each TVM. Using the TPM-Manager, the TVM is able to communicate with its vTPM.

Integrity Measurement Our security architecture supports integrity measurement facilities so as to provide an explicit statement about the configuration of the virtual machine. The management virtual machine provides a measurement agent, which computes a SHA1 measurement over the virtual machine’s software image before spawning the VM. The obtained measurements are stored inside the secure storage of the VM’s vTPM.

Complete Trust Chain For trust establishment in a remote entity, it is essential that the trust chain from the hardware-based trust anchor to the final application is not interrupted. Therefore, the TCG trusted boot process is extended further to the end-user application, including measurement of the bootloader, TVMM and VM.

Secure Storage To prevent unauthorized access to sensitive data, the security architecture provides a secure storage where data is protected by cryptographic measures. Access to this secure storage is only granted, if the authentication credentials are correctly asserted. The secure storage is protected by the TPM and, thus, through hardware measures.

Attestation Completeness Attesting the full image of a virtual machine allows a complete attestation of all software components, including all started processes, configuration files, kernel modules, and scripts. This property solves the problem of incomplete measurements, which was described in Section 7.3.

Compartmentalization The characteristic of platform attestation only enables making a statement about the trust level of a particular platform at a specific point in time. However, it cannot make a decision about future state transitions that can lead to a state that is not trusted. The TVMM must, therefore, possess mechanisms that only allow state transitions of the system that cannot influence the Trusted VM. This characteristic is achieved by the strong isolation property provided by a virtual machine monitor. This property solves inefficiency and time discrepancy problem that were described in Section 7.3. We will explain this issue in the next section.

7.5.2 Trusted Virtual Machine

Each virtual machine runs an own virtual appliance. Such a virtual appliance is usually configured to host only a single application (a secure transaction software) and the complexity of the included operating system is reduced to its minimal size. This approach significantly enhances system security, because the chance of vulnerabilities increases with the complexity of a system [15]. Consequently, the OS of the virtual appliance must be configured in such a way that it only supports a minimal number of user space software and kernel components, e.g., device drivers. It can, for example, be realized as a microkernel [102]. Device drivers have error rates that are three to seven times higher than ordinary code [35] and approximately 70 percent of the operating system consists of device drivers; therefore, reducing the total number of device drivers in the operating system clearly decreases the errors of an operating system, in turn, the potential vulnerabilities that could be used by an adversary to spy on a user's entered sensitive data. This fact is a basic design criteria for the Trusted Virtual Machine, which acts as a container to run sensitive software processes. If an adversary could exploit a vulnerability of a device driver, the adversary could spy on sensitive data entered by a user. In addition, the operating system running in the TVM must be configured in a way that it is not allowed to reach a state that is not fully trusted. Thus, it must be configured so as to be incapable of reaching a state that is not fully trusted. Thus, it must be configured to forbid the execution of additional software components, including kernel modules or user applications. Note that this approach is feasible since a Trusted Virtual Machine only consists of one specific application which is intended for a specific purpose. As a result, there is no need for executing additional software components.

In addition the OS, a virtual appliance contains one user application. This user application is a software application with a very high protection level, such as a secure homebanking client or a secure environment for displaying and signing documents with a smart card. Each time a new trusted VM is created, a virtual TPM instance is initiated and the PCRs 0-15 are filled with the PCR values from the underlying hardware TPM. Additionally, the hash value of the measured image is stored in the virtual TPM's

PCR[16]. The TVM also provides an attestation service (See Figure 7.2), which enables accessing the content of the platform configuration register and, thus, the secure reporting of its integrity to a remote entity.

To further potential reduce vulnerabilities the TVM distinguishes between program memory and data memory, similar to the Harvard architecture. The program memory holds the program machine code represented by a sequence of instructions and the data memory holds data that are related to the user's transactions. Any modifications to the program memory cannot be written back to the virtual appliance, meaning that each time before the TVM spawns, its program memory is reverted to its initial state.

This approach has the following advantage: The client software only has a single valid system state, which, in turn, reduces attestation complexity. The virtual machine is also equipped with a secondary disk image that can be used to store transferred data and information about former transactions. This data can be protected by storing the secondary disk image inside the secure storage provided by the Management VM. The data can also be protected by *sealing* it to the vTPM of the TVM, thus, directly binding it to a specific virtual appliance. We will discuss this security mechanism in Section 7.5.4.

Isolating program and data memory are realized through two different disk images. The secondary disk image is used to store state specific data. However, since data stored on the secondary disk image may be able to influence the runtime condition, only data that originates from the secure transaction software is stored there. Moreover, we also perform consistency checks to ensure that the disk image has not been compromised by malicious software. Because former transactions of the TVM are stored on a secondary disk image, the TVM can be easily updated without the loss of old transaction data, if newer version of the virtual appliance are published.

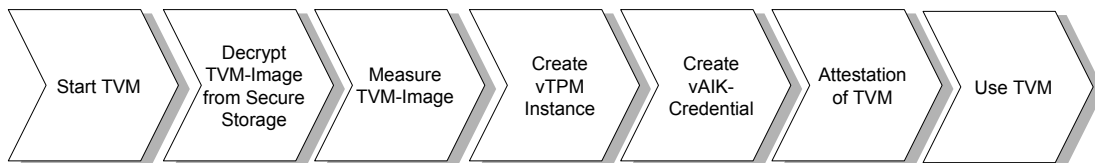


Figure 7.3: Actions taken when a Trusted VM spawns

The actions taken when a Trusted VM spawns are shown in Figure 7.3. The figure shows that before a trusted virtual machine is ready for use, a number of steps have to be performed that include the decryption of the virtual appliance (often also referred to as image) and the initialization of the TVM's virtual TPM. As soon as the TVM spawned, it can provide a proof about its trust level using platform attestation.

7.5.3 Virtual TPM

In order to successfully use a virtual TPM as trust anchor, the following requirements must be satisfied:

R.1: Performance This goal states that the overhead from using a TPM in a VM should be negligible compared to directly executing commands on the TPM. The VM should therefore be able to execute TPM commands at nearly the same speed as when these commands are used on a non-virtualized machine.

R.2: Compatibility It should be possible to execute TPM command code in a VM without modifying the code or adapting it to the virtualized environment. Therefore, we explicitly forbid using paravirtualization techniques [185].

R.3: Simplicity A fault in a VMM can cause a failure in all VMs which could result in a crashing VM. A VMM that provides abstraction and sharing of a TPM should therefore be as simple as possible [132].

R.4: Security A TPM that is used in a VM should provide the same security properties as if the TPM would be accessed natively.

R.5: Minimal modifications to the specification If the specifications by the TCG must be modified, these modifications should be as slight as possible, leaving the modifications and the specification as compatible as possible.

To satisfy these requirements, we are using a virtual TPM that is implemented in software and runs inside the Management VM. The advantage of a software TPM is that the requirements R.1, R.2, and R.5 can be achieved very easily, while the requirements R.5 and R.3 are harder to achieve. To achieve R.5, the software TPM needs to be protected by the hardware TPM to enable storing of persistent data. R.3 is partly achieved by implementing a software TPM that is as simple as possible.

In order to place trust into measurements reported from a virtual TPM, a complete trust chain from the hardware-based trust anchor up to and including the end application needs to be established. This includes all measurements performed by the hardware TPM as specified by the TCG [170], the bootloader (e.g., TrustedGrub [8]) and the hypervisor and the VM instances, including the virtual appliances. When a virtual TPM spawns, its PCR values are initialized with values from the underlying hardware TPM, as shown in Figure 7.4.

PCR	Content of TPM	Content of vTPM
0..7	Measurmnt. of CRTM and BIOS	Measurmnt. of CRTM and BIOS
8..15	Measurmnt. of the Bootloader and TVMM	Measurmnt. of the Bootloader and TVMM
16	empty	Measurmnt. of the virtual appliance
17..	empty	for free use

Figure 7.4: Mapping of the PCR values

In order to report the platform configuration of a TVM, a strong binding between vTPM and TPM must exist. Otherwise, it would be possible for the vTPM to report PCR values to a remote entity that are different from the ones that were measured by

the underlying hardware TPM. Berger et al. [20] have already proposed three different solutions for this problem. After careful considerations, we decided to employ the solution in which each virtual AIK is bound to a hardware AIK. We believe that this solution has several advantages over the others. On the one hand it has strong similarities to the concepts provided by the TCG specifications as an additional Privacy-CA, and on the other hand, it allows quick spawning of additional vTPM instances.

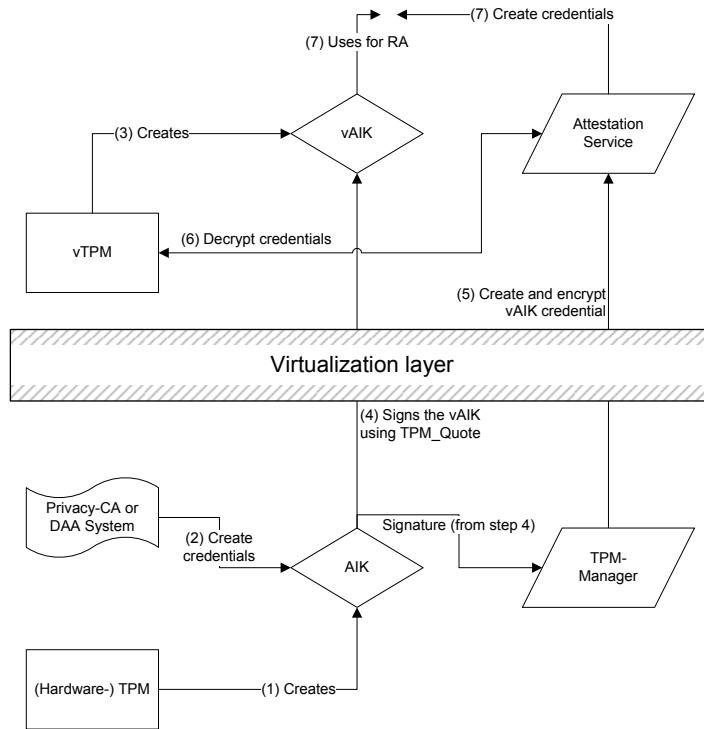


Figure 7.5: Binding the vTPM to TPM

Figure 7.5 shows the association of a vTPM with an underlying hardware TPM, as realized by our architecture. In step 1, the host TPM creates an AIK and in step 2 retrieves a corresponding credential through a Privacy-CA or via DAA [23]. In step 3, the vTPM then creates a *vAIK*, which is signed by the Host TPM, using his own AIK. To prevent replay and masquerading attacks, a *Nonce* provided by the vTPM instance, an additional *timestamp*, and the hash values of the hardware TPM's PCR[0..15] are embedded into this signed message. The integration of the PCR value in the *vAIK* credential is necessary, since a trusted software configuration could otherwise be forged. In Section 7.7.2, we will discuss why this concept prevents an attacker from forging a trusted software configuration. The corresponding credential is then encrypted using the K_{vEK} and sent to the vTPM instance (5), that is in possession of K_{vEK}^{-1} and, therefore, able to decrypt this message. A *vAIK* credential contains the vTPM generated public vAIK and consists of:

- Timestamp
 - Hardware PCR[0..15]
 - Nonce
 - *vAIK*
 - AIK credentials
- } signed with *AIK*

For the remote attestation process, the *vAIK* is used to perform the `TPM_Quote` command. The verifying party can then decide whether the remote platform configuration is trusted by validating the *vAIK* credential and the delivered output of the `TPM_Quote` command. A platform configuration is deemed trusted if the following conditions are satisfied:

- The *vAIK* is authentic and generated by a vTPM
- The vTPM is authentic and protected by a hardware TPM
- The hypervisor including management VM is in a trusted system state
- The hardware TPM is authentic
- The TVM is in a trusted state

7.5.4 Secure Storage

Secure storage is an isolated memory area where sensitive data is stored. Access to this area is always controlled by the management VM and only authorized processes are allowed to access this storage. The secure storage provides integrity, confidentiality and authenticity and is protected by non-migratable keys, which reside in the hardware TPM. It is not resistant against replay attacks, where a user replays the actual secure storage with an old secure storage. To achieve protection against attacks of this type, some sort of TPM protected counters, as introduced in [137] or [160] could be used. However, we do not consider these attacks, since they cannot be conducted by malware and are only relevant in the Information Rights Management scenario. In addition, these solutions do not enable data backup, thus, creating the need for an additional backup concept. We use the secure storage for the following three purposes:

Protecting State Specific vTPM Data

Since a virtual TPM is a software TPM, it is necessary to protect state information of the vTPM with hardware measures. For this purpose, each vTPM instance possesses its own memory structure, where vTPM state specific data, such as cryptographic keys, is stored. To prevent an adversary from modifying this structure, the whole structure is either encrypted or sealed with a non-migratable key that resides inside the hardware TPM. When a virtual machine and its associated vTPM instance spawns, the TPM-Manager (See Figure 7.2) decrypts the memory structure of the vTPM and assigns the decrypted vTPM memory structure to the currently spawned vTPM instance.

Protecting VM Images

Every virtual appliance of a virtual machine can be stored inside the secure storage. The virtual appliance is then protected by a non-migratable key stored inside the hardware TPM. Before a specific TPM-protected virtual machine can be spawned, the TPM-Manager decrypts the image using the non-migratable key stored inside the hardware TPM and launches a VM using the decrypted image. This concept prevents an adversary from maliciously modifying a virtual appliance without the notice of a particular user.

Protecting VM Specific Data

Sensitive transaction data can be stored inside the secure storage of the Management VM. This is realized by a combination of security services offered by the Management VM as well as VM specific security services. The state specific vTPM data is protected by a security service running in the Management VM, and the transaction data of a specific VM is protected by a security service running inside a specific VM.

Protecting VM specific data can again be realized by either sealing the data to a specific state or by encrypting the data with an encryption key that resides in the virtual TPM. The choice depends on the scenario and is implementation specific. In the following, we will show, in detail, how data can be protected by sealing it. This concept provides better security compared to only encrypting data; this is due to the fact that state changes (for example changes caused by malicious modifications to the hypervisor [93]) are also detected. However, protecting data by encrypting it with an encryption key that resides inside the TPM is analogous and requires only minimal modifications to the mechanisms presented here. For simplicity reasons, we will in the following only make use of asymmetric cryptography. However, to enhance efficiency a hybrid encryption scheme could also be used. In that case, the storage data is encrypted with a symmetric key and the symmetric key is bound to the TPM using an asymmetric key.

We denote the structure that stores transaction data with D . This structure is then sealed by a VM specific security service to a specific time Δt using the following function: $Seal(D_n, PCRInfo, H_{vSRK})$. H is a key-handle for the relevant storage root key and PCRInfo is a TPM_PCR_INFO structure that contains the information to which PCRs D_n will be bound. The cryptographic key used to unseal the structure D is the storage root key $vSRK$ of the virtual TPM. The TPM_PCR_INFO structure [168] contains the PCR values 0-16 and, thus, a specific vTPM state. The operation to unseal is denoted $Unseal(\{D_n\}_{vSRK})$ where for simplicity reasons $\{D_n\}_{vSRK}$ also includes the structure of the platform configuration registers. Unsealing the $\{D_n\}_{vSRK}$ at $\Delta t + x$ is then only possible if the current platform configuration, i.e., the PCRs of the vTPM, is equal to the platform configuration that $\{D_n\}_{vSRK}$ is bound to.

Using the Management VM to spawn a Trusted VM that provides TPM protection to transaction data, therefore, requires a combination of sealing data to a vTPM and sealing data to a hardware TPM. Figure 7.6 shows the involved protocol steps.

1.	MVM \rightarrow	TPM	: TPM.OSAP(<i>authData</i>)
2.	MVM \leftarrow	TPM	: H_{SRK}
3.	MVM \rightarrow	TPM	: $Unseal(\{vTPM_Storage\}_{SRK}, H_{SRK})$
4.		TPM	: Check if the values of the current PCRs [0–15] are equal to the PCRs $\{vTPM_Storage\}$ is bound to
6.	MVM \leftarrow	TPM	: $vTPM_Storage$
7.	MVM \leftarrow	TPM	: TPM.SUCCESS
8.	MVM		: Initialize vTPM
9.	MVM		: Launch and measure TVM
10.	TVM \rightarrow	vTPM	: TPM.OSAP(<i>authData</i>)
11.	TVM \leftarrow	vTPM	: H_{vSRK}
12.	TVM \rightarrow	vTPM	: $Unseal(\{D_n\}_{vSRK}, H_{SRK})$
13.		vTPM	: Check if the values of the current PCRs [0–16] are equal to the PCRs D_n is bound to
14.	TVM \leftarrow	vTPM	: D_n
15.	TVM \leftarrow	vTPM	: TPM.SUCCESS

Figure 7.6: Protocol steps of a spawning vTPM that is protected by the hardware TPM

First, a TPM Object-Specific Authorization Protocol (OSAP) session is created between the TPM and the MVM. This command requires authorization (owner-password and SRK -password) and computes a cryptographic secret and a handle (H_{SRK}) to this session; both to protect the whole session traffic (see [174], pp. 71). The returned handle is then used to issue the *Unseal* command. The TPM then unseals the encrypted $vTPM_Storage$ and delivers the output back to the MVM. The MVM then initializes a new vTPM as shown in 7.5 with the decrypted $vTPM_Storage$ and spawns and measures a new TVM.

7.6 Attestation of Virtual Machines

To enable the trusted VM to report its integrity and, thus, to place trust into the used system configuration, attestation techniques are used. For this purpose, we use the Protocol 3.6.3, which establishes a secure attestation channel and prevents masquerading attacks on the authenticity of the platform configuration. In addition, it guarantees an end-to-end communication and prevents the attestation channel from becoming compromised by another component of the security architecture (e.g., Management VM) that could take over the attestation channel after the attestation has succeeded.

In our approach, the attestation service is located inside the TVM. This is different than [137, 87, 86] where the attestation service is part of the security kernel. However, our approach has two advantages over integrating the attestation service into the security kernel (in our case the Management VM):

- (1) It enables reducing the complexity of the security kernel (Trusted Computing base), since only the necessary components for integrity reporting, such as the measurement agent, are integrated.
- (2) It enables supporting a wide-variety of different attestation methods each satisfying different security objectives.

The second advantage is of vital importance since it seems unlikely that only one unique attestation method will be used by distributed applications that use remote attestation. Integrating the attestation service inside the security kernel would require to adapt the attestation service and to integrate all attestation services inside the security kernel. This is a potential vulnerability that is exacerbated by the fact that the attestation service runs in the highest privilege-level of the CPU.

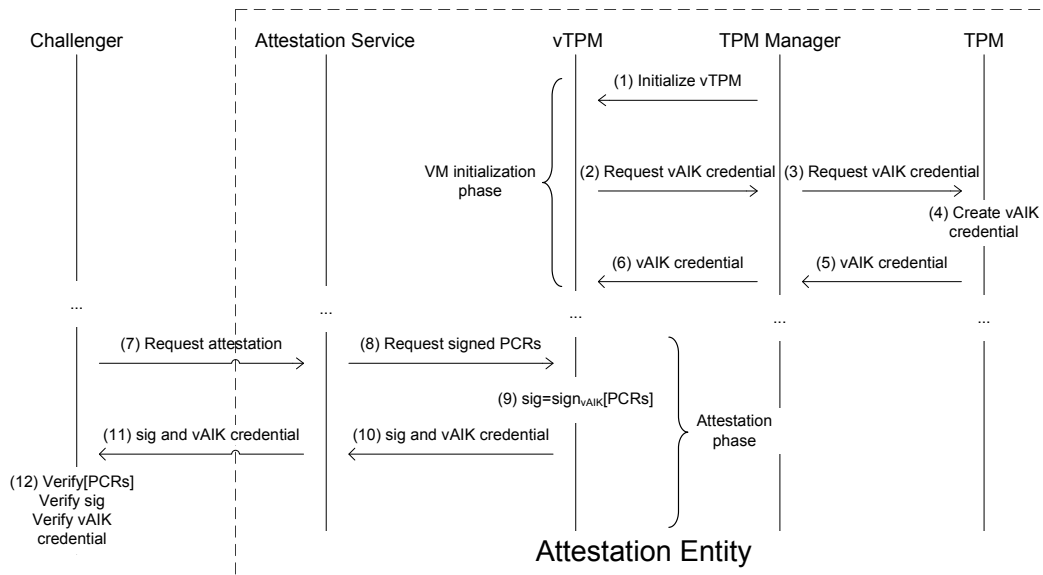


Figure 7.7: Simplified attestation process

The simplified process of our remote attestation protocol is illustrated in Figure 7.7. It consists of an initialization phase and an attestation phase. The initialization phase yields a *vAIK* credential, which is then used in the attestation phase to sign the PCRs. This *vAIK* credential is generated and signed by an AIK from the hardware TPM as explained in Section 7.5.3 and shown in Figure 7.5. In the first step of the initialization phase, the vTPM is initialized, which, in turn, requests a new *vAIK* credential from the hardware TPM (steps 2 and 3). The hardware TPM issues a *vAIK* credential and sends it to the vTPM (steps 5 and 6). The attestation phase then establishes a secure attestation channel as explained in Section 3.5.

7.7 Evaluation of the Architecture

In the following section, we evaluate our proposed architecture. We will first divide our architecture into layers and show how each layer provides protection mechanisms for the layers above. Using this approach, we show how we solve the security challenges described in Section 7.2.

7.7.1 Protection Layers

The protection architecture is shown in Figure 7.8 and consists of components divided into four protection layers. Each component located on a single layer provides security mechanisms to protect the components located on the layer directly above. In the case of a successful attack on one layer, the layers below can prevent the attacker from successfully accessing confidential data.

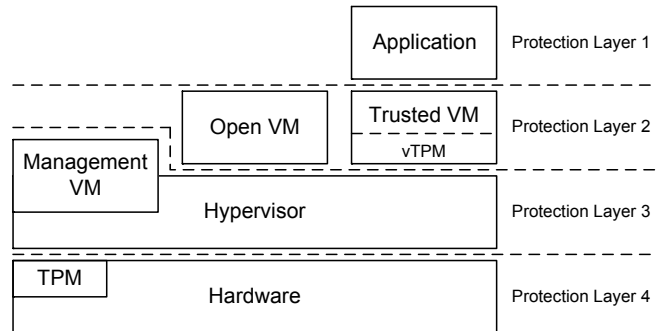


Figure 7.8: Protection architecture organized in layers

Protection Layer 4: TPM and Hardware

The TPM is the anchor of trust and the basis for the attestation. It holds several non-migratable, client-specific keys to protect the vTPM and the secure storage of the security architecture. The protection mechanism of this layer is twofold: it provides a mechanism with which each executed software component can be identified (1) and a secure and protected storage (2).

Such a protection mechanism prevents attacks on the authenticity of the software components, for instance, an attack where some sort of subversive software component tries to foist a user into entering sensitive data into a compromised software application. Such an attack can, for example, be conducted if a user accidentally executes a malicious software component that modifies parts of the operating system or installs a malicious hypervisor [93] to enable spying on sensitive data. Using characteristics of virtualization-supporting hardware, such attacks can also be performed without rebooting the system [135]. These attacks are also relevant if a system is used that is based on virtualization as shown in [93, 192, 187]. This shows that virtualization techniques on their own are not sufficient to prevent attacks on the authenticity of software

components, but require attestation techniques that enable detecting whether a system has been compromised.

Protection Layer 3: Hypervisor and Management VM

The hypervisor is the first part of protection layer 3 and provides an abstraction layer for the underlying hardware. It provides the protection mechanisms discussed in Section 7.5.1, which are: strong isolation of the virtual machines (i.e., compartmentalization), attestation, integrity measurement, a complete trust chain and attestation completeness. These mechanisms ensure that different virtual machines cannot influence each other, e.g., by reading each other's memory. Each virtual machine uses a virtual device driver to communicate with the underlying hardware. The hypervisor ensures that these device drivers can only access the memory of the corresponding virtual machine. In contrast, when applications of different trust levels are running on a machine without virtualization, an attacker could use a malicious device driver to gain system wide access (Compare Section 7.2).

The management virtual machine is the second part of protection layer 3. It is responsible for starting, stopping and configuring the virtual machines. It is included in protection layer 3 because it is closely connected to the hypervisor and is a privileged VM with direct access to the hardware TPM.

Protection Layer 2: Open VM and Trusted VM

Protection layer 2 consists of the open VM and the Trusted VM. The open VM is allowed to run arbitrary software components. It runs applications with a lower trust level, such as web browsers or office applications. The open VM provides the semantics of today's open machines and, therefore, has no additional protection mechanisms for upper layers.

The TVM runs the trusted transaction client and a tiny OS or a microkernel with a minimal number of software components, to reduce the possible number of security vulnerabilities.

The TVM runs in protection layer 2 and provides a virtual TPM as an additional protection mechanism. The operating system of the TVM uses this vTPM to protect the trusted transaction client running in protection layer 1. We use this vTPM to perform a complete attestation of the entire hard disk of the TVM, thereby ensuring that neither the operating system, its configuration, nor the attestation service running on top of the operating system are manipulated. Note that this is only practical when a minimal operating system with only one special-purpose application is used. As a result, the verification of the state of the entire virtual machine requires only one reference value. This eliminates the main disadvantage of the binary attestation, namely the need to maintain a large amount of reference values. Moreover, during the attestation, a verifying entity checks whether the protection layers below the operating system, e.g., the hypervisor, are trusted. If the TVM establishes a secure channel to another remote entity, the vTPM is additionally used to ensure authenticity of software integrity values

and to protect cryptographic secrets. The corresponding protocol is discussed in Section 7.6.

All I/O interfaces that can be used to extract data from a TVM need to be blocked or controlled. For example, the management VM ensures that the program and data memory of the TVM are isolated and the TVM cannot modify the program memory. This mechanism prevents an attacker from modifying the program memory and booting into a malicious platform configuration.

Protection Layer 1: Secure Transaction Client

Protection layer 1 consists of the application that enables secure transactions with arbitrary platforms. In order to enable secure transactions, protection layer 1 enables establishing secure channels using the vTPM as well as answering attestation requests. It is implementation specific and whether or not the transaction client provides additional security mechanisms, such as mechanisms to harden it against runtime attacks, depends on the application scenario. We will show in Chapter 9 how such a secure transaction architecture can be realized.

7.7.2 Security Evaluation

In the following section, we evaluate and analyze whether the security mechanisms described in Section 7.7 solve the security challenges described in Section 7.2. We assume that the secure transaction client, which we introduced in Section 7.2, is running in a TVM and that its integrity is ensured using attestation techniques. Based on the attacker model introduced in Section 7.2.2, we consider an attacker who tries to access sensitive transaction data.

Software Manipulation Attacks

Each protection layer is measured before execution by the layer below. This results in a chain of trust with the property that all software manipulation attacks that modify a software component before it has been measured are reliably detected. However, to prevent the detection of a compromised system configuration, an attacker can use the following attack patterns:

- (1) he manipulates the software integrity after it has been measured using a runtime attack,
- (2) he alters the software integrity measurements so that they reflect a trusted software configuration, and
- (3) he manipulates the measurement process so that a specific subversive application is not measured or the chain of trust is not complete.

In the following, we will study different underlying attack vectors that an adversary may use in order to successfully perform an attack of types (1), (2) and (3). Figure

7.9 shows these possible attack vectors. The arrows 1-4 indicate a starting point of a possible attack vector and the arrows that additionally possess an alphabetic character mark the further proceeding of an adversary.

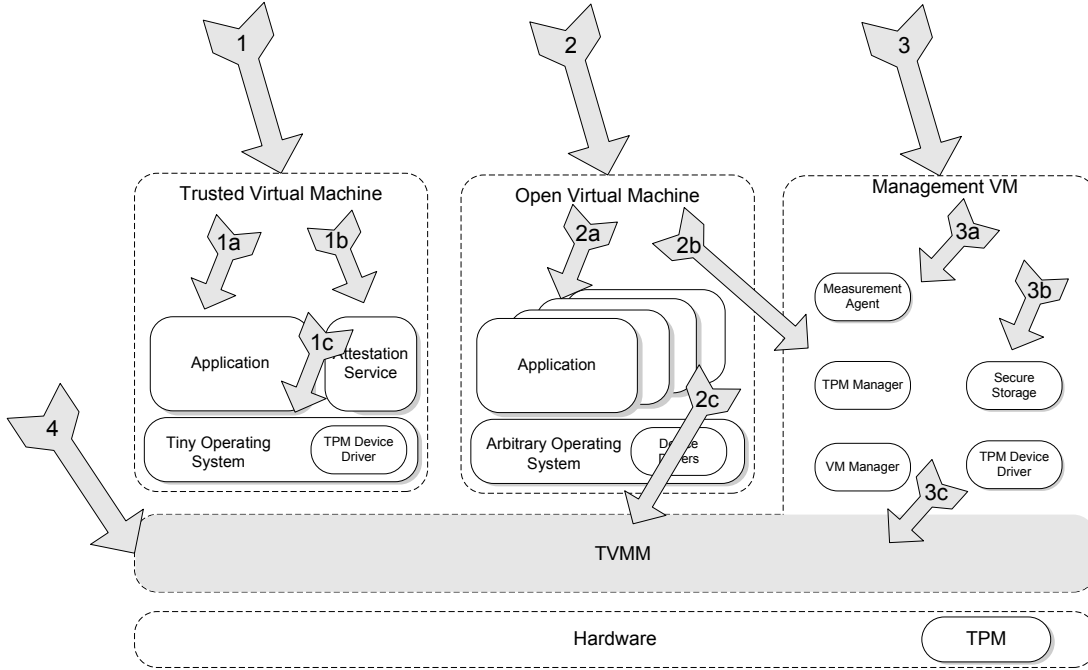


Figure 7.9: Attack vectors on our security architecture

Attacks on the TVM (Arrow 1) The attacker may be able to exploit a vulnerability of one software component in order to perform an attack of type (1). The risk of these runtime attacks on the TVM's operating system is reduced, because the operating system of the TVM is directly adapted to only host one user application and has a small code complexity. This reduced complexity enables the use of a very simple operating system with only a small set of services. Since simplicity was found to be the source of greatest protection against system penetrations [9], this approach is more resistant to runtime attacks than using a more complex operating system. Moreover, we choose a strict system configuration to minimize possible attack methods, e.g., network connections are restricted, the hard disk of the TVM is read-only and swapping is disabled. Since the TVM's operating system is either realized as a microkernel, such as MINIX [79] or L4 [101], or it has the characteristics of a microkernel, it is basically possible to formally verify its correctness w.r.t. certain precisely stated security predicates. Thus, formal methods [95] could be used to ensure that no exploitable vulnerability exists, thereby resulting in verified security [124]. Additionally, the OS is configured in such a way that it is not possible to execute additional software components with the privileges of the operating system. For example, a static security kernel is used that not only

prohibits the dynamic reloading of modules, but also all other possibilities to access the whole memory allocated to the VM (e.g., file systems such as `/proc/kcore` has to be disabled). Attacks of types (2) and (3) cannot be performed since the processes needed to be manipulated in order to perform a successful attack are located on the layers below.

Attacks on the Open VM (Arrow 2) The Open VM is allowed to run arbitrary software components and provides the semantics of today’s open machines. As a result, the environment consists of multiple different applications with a wide spectrum of security vulnerabilities. As a consequence, it is very likely that an attacker is able to perform an attack of type (1) by exploiting a vulnerability to gain control and install subversive applications. In contrast to the TVM, the Open VM is not equipped with a means of measuring software components. Therefore, it is not possible to detect that the software environment has been maliciously modified. However, since the underlying protection layer provides isolation of software environments, it is not possible to access sensitive data, since security breaches by compromised applications are contained inside the Open VM. Thus, it is not feasible for an attacker to access sensitive data stored inside the TVM or inside the secure storage. However, since it seems likely that the attacker is able to gain full control of the Open VM, he may be able to perform a hardware attack and break out of the compartment, thereby undermining the isolation characteristic of the hypervisor. These attacks are visualised by arrows 2b and 2c of Figure 7.9.

Assume that an attacker has completely taken control of the Open VM. He might then try to perform a malicious DMA access to a memory area allocated by the Hypervisor, Management VM or Trusted VM. For example, the very widespread Xen hypervisor [14], which we also use in our approach, allocates the top 64MB of each address space. Under normal conditions, this address space is protected by the CPU and the operating system of the Open VM cannot directly access this area. However, since DMA bypasses any protection managed by the CPU, an attacker can use the physical address of the hypervisor’s allocated address space to access the address space and then overwrite key-data structures. [187] and [53] recently showed that the Xen hypervisor is vulnerable to such attacks and that an attacker can circumvent the isolation characteristic of a hypervisor. Using this approach an adversary can access the hypervisor, thereby giving him access to the memory of all VMs since the hypervisor is privileged against all VMs. We will further analyze this attack in Section 7.7.2 and also present a mechanism to cope with this threat.

Attacks of type (2) and (3) can be performed if the attacker has taken full control of the system and, thus, successfully broken out of the Open VM. Since the attacker then has the same potential as if he had successfully overtaken the Management VM directly, we will discuss this issue in the next paragraph.

Attacks on the Management VM (Arrow 3) The Management VM runs a tiny operating system or microkernel with only a small set of services. These services are

mainly the security services explained in Section 7.5.1, namely the *VM Manager*, *TPM Manager*, *TPM Device Driver*, *Secure Storage* and a *Measurement Agent*. In addition, the Management VM provides split-device drivers [62] for all devices that cannot be virtualized with hardware measures [37]. A split-device driver is necessary for devices such as the TPM since currently available TPMs cannot be virtualized with hardware measures, as we will explain in Chapter 8.

To successfully perform an attack of type (1), an adversary has to exploit a vulnerability in either the operating system of the Management VM, or in any of the security services offered by the Management VM. Since the memory of the management VM is also isolated from the memory of the TVM, a hardware attack, as explained in the preceding paragraph, can be used to access the memory of a TVM. This is visualised as arrow 3c in Figure 7.9. The risk of such runtime attacks on the Management VM are reduced, because the code complexity of the Management VM is very low and software with lower complexity is expected to have less errors than software with higher complexity. To additionally enhance the security, formal methods should be used to prevent the emergence of software vulnerabilities that could be exploited by an adversary. However, another disadvantage is that our software TPM extends the code base of the management VM, thus violating R.3 of a virtual TPM.

Additionally, the Management VM is responsible for providing a Secure Storage and the Measurement Agents. If an attacker gained access to the Management VM and is not able to perform a malicious DMA to access the memory of a TVM, he might perform an attack of types (2) and (3). He could achieve a type 3 attack, for example, by substituting the virtual appliance of the Trusted VM with a malicious virtual appliance and altering the measurement agent so that the agent provides malicious measurements (attack (3)). Alternatively, he could achieve an attack of type 2, for example, by altering measurements stored inside the virtual TPM's storage or accessing cryptographic keys stored inside the vTPM. However, to successfully perform these attack types, he needs to modify the security services during runtime, i.e., after they have been measured. This is necessary since during the attestation, not only the measurements of the vTPM are validated, but also the measurements of the underlying hardware TPM. If an adversary only modifies the measurements of the vTPM, the measurements would differ from the measurements stored inside the hardware TPM (compare Section 7.5.3). However, it is much easier for an adversary to access data which are stored in a virtual TPM than if they were stored in the hardware TPM. For example, it is possible for an adversary to extract keying material from a software TPM by accessing the memory during runtime. Although this may result in a malicious software configuration which can be detected, an adversary might have gained access to sensitive keying material. As a result, our software TPM does not completely achieve the security requirement for a virtual TPM (R.4) as presented in Section 7.5.3.

Attacks on the Hypervisor (Arrow 4) The hypervisor consists of a very low code complexity (e.g., Secvisor comprises of only 1112 lines of code [148]), particularly when compared to operating systems, and does not expose interfaces that could be used for

an attack; therefore, attacks of type (1) are difficult. If an attacker were to gain access to the hypervisor, he could perform attacks of types (1) and (2) as already discussed in the preceding paragraph.

Manipulate Input

Input manipulation attacks try to access confidential data by capturing keystrokes. In our security architecture, these attacks can be situated on the TVM, the Management VM or the hypervisor.

Manipulating the input at the TVM is rather difficult since the TVM's code complexity is very low and the only possibility would be to perform a runtime attack and then install some sort of key-logger. Since the integrity of the TVM is measured each time it spawns, the time window to perform such an attack is very small. In addition, each attack needs to be performed from scratch since it is not possible to persistently modify the program code of the TVM. Another possibility for spying on a user's entered confidential data is to modify the hypervisor or the management VM. This would lead to a successful attack if the keyboard device-driver were virtualized using software techniques [62]. If software techniques have been used to virtualize the keyboard device driver, a backend device driver resides in the Management VM and can be altered to spy on a user's entered data. In that case, a successful runtime attack on the Management VM or the hypervisor must be performed.

However, if the keyboard device-driver are virtualized using hardware techniques [37], only a successful runtime attack in the TVM enables an attacker to capture keystrokes. Gaining access to the hypervisor or Management VM would in the latter case not be sufficient to capture keystrokes.

Manipulate Output

Output manipulation attacks manipulate the information output of a software system. These attacks are prevented by ensuring that the software integrity of the devices for information output are trusted. Here, as in the preceding sections, runtime attacks are still possible and the only threat.

Output attacks that exploit human characteristics, such as phishing [44], can be easily prevented by binding transaction data directly to a specific transaction session using cryptographic keys; because this solution is implementation specific and depends on the application scenario, we will present the security mechanisms in Section 9, where we present an application scenario and the corresponding security mechanisms.

Hardware Attacks

We have already discussed that hardware attacks can be used to perform software manipulation attacks. One such critical hardware attack is a DMA attack, currently the only method to circumvent the isolation characteristic provided by the hypervisor.

DMA attacks can either be handled entirely in software by emulating all I/O devices, thereby causing a high performance degradation, or by providing a hardware I/O MMU

[186, 19] that supports secure DMA, e.g., as provided by Intel’s Trusted Execution Technology [38] or by AMD’s SVM [6]. A hardware I/O MMU has similar characteristics to a conventional Memory Management Unit (MMU) and translates VM-visible virtual addresses used in DMA transactions to physical addresses. It also provides memory protection against illegal access by I/O devices.

These hardware mechanisms, thus, can prevent a virtual machine from breaking out of its compartment by using malicious DMA accesses. As a consequence, our security architecture can prevent malicious DMA accesses if a hardware I/O MMU is used. Probing attacks are difficult to inhibit, but we do not consider them, because they cannot be performed by software components.

7.8 Implementation

The security architecture has been implemented in Java using version 3 of Xen [14]. Further details on the implementation can be found in [81]. The Xen hypervisor is very compact and, therefore, fulfills the requirement of a small Trusted Computing base. It employs a unique VM called Dom0, which is created in the initialization phase and is responsible for spawning new VMs. This Dom0 component, therefore, acts as the Management VM in our architecture. It is also responsible for the assignment of I/O devices to the VMs. Other VMs spawned by the Dom0 are called DomUs (User domains) and run a paravirtualized Linux. The DomUs utilize the driver support of the Dom0 by introducing a *split device driver* [62, 7]. This split device driver, as the name implies, is split into a back end and a front end. The back end runs in the Dom0 and the front end is implemented in the DomU. Since the Dom0 has privileged access to the other DomUs, it needs to be trusted. Therefore, we suggest that the Dom0 only runs a minimal operating system, while the open VM, which is realized through another DomU, runs the productive operating system. Our implemented security architecture runs on a IA-32 Core2 Duo Processor, which supports the preliminary VT-d architecture [37] and, thus, an I/O MMU as well as the establishment of DRTMs. However, at the current stage of implementation, we do neither use an I/O MMU nor a DRTM.

Our implemented security architecture is shown in Figure 7.10, which displays one virtual machine, the hypervisor, Dom0, the different vTPM parts, and the communication paths between those components and the underlying hardware TPM.

Modification of vTPM

The available vTPM module in Xen has been extended to allow integrity measurement. The vTPM module is a reduced driver-pair compared to the one introduced in [20], and, therefore, neither supports attestation nor migration of vTPMs. The modifications to the back end vTPM (vTPM Manager) and the front end vTPM interface required nearly 300 lines of code. The vTPM and vTPM manager were extended with the commands *TPM_CreateVTPMCredentials*, *TPM_RetrieveVTPMCredentials* and *TPM_VTPMManager_ResetSessions*. *TPM_CreateVTPMCredentials* enables a vTPM to retrieve a *vAIK* and an *EK* and send them to the back end vTPM

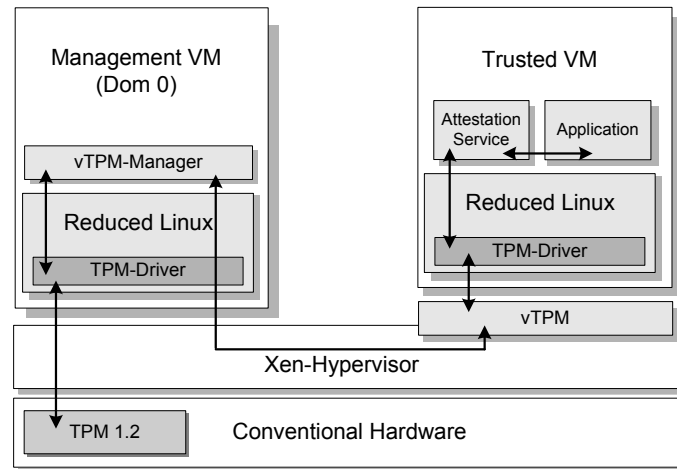


Figure 7.10: Implemented Architecture using Xen. The arrows indicate the communication path between the different components.

manager. The vTPM Manager then uses a Java Host Tool to communicate with the hardware TPM which, in turn, carries out the hardware operations.

The client application was developed by extending the already existing Java framework² to allow spawning of several virtual TPM instances. The main advantage of the tpm4java framework over *trusted Java*³ is that it is independent of *TrouSerS*⁴, and can, therefore, talk directly to the `/dev/tpm` device. The library was also extended to support the issuing of *vAIKs* by adding the TPM commands *TPM_CreateVTPM-Credentials* and *TPM_RetrieveVTPMCredentials*.

Integrity Measurement

For integrity measurement, we replaced the existing starting scripts of Xen with a wrapper that loads the image of the virtual machine from the secure storage, extracts the image, and calculates the hash of the image. The hash value is then temporarily cached in the Dom0 and the virtual machine is started with the extracted image and a second empty disk image for storing application specific data. This allows us to store arbitrary content, e.g., information about former transactions, without influencing the status of the VM. The second disk image can also be protected by either binding it directly to the underlying TPM or by sealing it to specific platform configuration registers as explained in Section 7.5.4. The vTPM-Manager then transfers the PCRs of the underlying hardware TPM and the measured image to every spawned vTPM instance. In our current implementation, the spawning of an additional trusted VM with a 2GB image takes approximately 124 seconds since a new *vAIK* credential has

²<http://tpm4java.datenzone.de/trac>

³<http://trustedjava.sourceforge.net/>

⁴<http://trousers.sourceforge.net/>

to be generated by the TPM. This also includes the calculation of the hash value of the disk image, which is the main source of processing time. The VM pauses in an idle phase until the *vAIK* credential has been issued. The vTPM is then initialized as shown in Figure 7.4.

Attestation Protocol

An attestation service runs inside the Trusted Virtual Machine. This attestation service awaits attestation requests from a remote entity and answers the requests by invoking the virtual TPM. The virtual TPM is in possession of a *vAIK* credential that has been issued by the underlying hardware TPM. Using this credential, a TVM is able to prove its trust level to a remote entity without invoking components of the underlying hypervisor or Dom0. The attestation service is completely implemented in Java and its integrity is ensured by validating the measured hash-value of the TVM image.

7.9 Related Work

Another approach that enables attestation is used in the Integrity Measurement Architecture (IMA) [142], where the authors present a comprehensive prototype based on Trusted Computing technologies and wherein integrity measurement is implemented by examining the hashes of all executed binaries. However, the prototype is not based on virtualization technologies, and, therefore, achieves no strong isolation between processes. This makes it necessary to transfer a complete SML to the remote entity, which, in turn, must validate all started processes to determine the trust level of a platform.

Berger et al. [20] illustrate how to virtualize a TPM and present a driver-pair that utilizes these concepts. Our work differs in the attestation process of virtual machines and in the way the mapping between the PCRs is performed. In contrast to [20], where the IMA is used for providing measurements, we argue that a complete verification is reached by only measuring the image of a VM. Furthermore, Berger et al. do not specify how the binding of a virtualized TPM (vTPM) to a TPM is performed.

Mccune et al. [107] propose an architecture that allows the execution of a completely isolated application without requiring a VMM. The approach utilizes a hardware-based dynamic root of trust provided by AMD's SVM architecture [6] or Intel's TXT architecture [38]. Using these hardware architectures, Mccune et al. execute a completely isolated software component directly on the hardware. The authors propose to only execute a very small piece of application logic (PAL) in this environment. This approach could also be adapted to our security architecture and the TVM could be executed directly on hardware within such a PAL. However, since this approach does not support virtual TPMs, the secure storage and the attestation mechanisms of the TVM must then be realized in another way.

Numerous approaches have taken shape that try to construct secure operating systems or secure operating system environments. These approaches can be divided into two categories (*Security Kernels* and *Virtual Machine Monitors*).

7.9.1 Security Kernels

The first category covers approaches that try to develop new security kernels from scratch. Approaches that can be included in this category are all approaches that are based on a microkernel, such as L4 [101], Nizza [154], Minix [78, 80] or Exokernel [58]. All these approaches provide only a minimal security kernel that only provides a small set of services, such as Inter-Process Communication and Memory Management. Device drivers are running in a higher privilege-level of the CPU, often referred to as *Server-Domain*, and are not part of the security kernel. Security kernels do not necessarily enable establishing multiple isolated operating system environments. However, they often provide a stronger isolation between processes. All these approaches try to construct secure and reliable operating systems. However, they do not integrate Trusted Computing functionalities to ensure that a system behaves as expected. This gap is expected to be filled by the Open-TC project [110] and the EMSCB-project [55]. Both projects try to enhance these approaches with Trusted Computing functionality.

7.9.2 Virtual Machine Monitors

The second category covers approaches that try to establish secure operating system environments. These environments are characterized by the fact that they do not necessarily require new operating system design, but use many functionalities of existing operating systems. All approaches are based on a virtual machine monitor fall into this category. Microkernels and virtual machine monitors have many things in common [76, 77]; however, microkernels are basically more complex than virtual machine monitors. Virtual machine monitors only provide abstraction to the hardware and partitions the hardware. In contrast to microkernels, it is not possible to execute applications directly on this virtual machine monitor. Thus, these environments require one or more operating system to execute additional applications. The following proposals try to establish secure operating system environments:

Garfinkel et al. [67] introduce an approach to create a virtual machine-based platform that allows attestation of virtual machines called Terra. Garfinkel et al. utilize VMWare's ESX Server to establish different types of virtual machines (*Closed Box* and *Open Box*) and to report the state of a closed box machine to a remote entity. Our approach differs in that Terra neither uses a hardware-based trust anchor (the TPM in our approach) nor allows attestation without the direct involvement of the VMM. For attestation, the VM in Terra has to contact the attestation interface that is being executed on the VMM, which can only report the status of the VM at the time of initialization. Using a virtualized TPM, in contrast allows, on the one hand, direct attestation and the reporting of fine-grained platform configurations and, on the other hand, the full functionalities of a trusted platform module, e.g., sealing. Terra also suffers from using a large Trusted Computing base, which is a potential security threat and is, therefore, not applicable in high security environments.

Palladium formerly known as Microsoft's Next Generation Secure Computing Base (*NGSCB*) [56, 118] is an approach that aims at establishing a small Trusted Computing

base, while satisfying the need for open mass-market operating systems. The approach is also based on a virtual machine monitor, called an *isolation kernel*, which establishes two different execution environments, i.e, VMs, with different trust-levels. NGSCB highly depends on hardware virtualization technology as well as on Trusted Computing technology. NGSCB also requires a security kernel (nexus) that runs on the isolation kernel. This nexus has similarities to the TVM in our approach. However, it does not provide any means of using attestation techniques for trust-establishment.

Secvisor is another very interesting approach that has been proposed by Seshadri et al. [148]. In this approach, the authors propose a very small virtual machine monitor that heavily makes use of virtualization-supportive hardware. Since Secvisor only supports one virtual machine, the hypervisor does not need to perform scheduling or interrupt handling and is, thus, very small. Secvisor aims at ensuring code integrity for commodity OS kernels. However, since it only supports one virtual machine, isolated security sensitive environments that can be used for sensitive transactions can hardly be formed.

Jansen et al. [87] also showed how the integrity of different virtual machines can be ensured by adding Trusted Computing technology to a virtual machine monitor. In this context, Jansen et al. make use of the Xen-hypervisor and, similar to us, add Trusted Computing to the Dom0. However, in contrast to our work, Jansen et al. propose to integrate the attestation service in the privileged Dom0. We believe that this approach is rather impractical since integrating the attestation service inside the Dom0 increases the complexity of this Domain. In addition, Jansen et al. only implemented parts of their design, and, therefore, it can only be speculated how the exact realization will look.

7.10 Summary

We have presented the design and security mechanism of a security architecture that is capable of supporting attestation techniques. Our proposed security architecture is based on virtualization-techniques and provides an isolated security environment where confidential data can be processed. We introduced a four layer protection architecture based on virtualization technology and attestation techniques. The four layer approach enables (1) leveraging attestation techniques for trust-establishment by achieving a meaningful attestation process and (2) containing security breaches in an isolated compartment. This approach guarantees software integrity and prevents an attacker from tampering with the software configuration of a platform. Our proposed security architecture can be used as a foundation for constructing next-generation secure operating systems that rely on Trusted Computing technology. Since we do not use some sort of sealing techniques, sytem updates in our architecture can be easily handled. However, the virtual TPM used in our approach is not able to achieve all our introduced security requirements for a virtual TPM. As a result, a new approach which we will discuss in the next chapter is necessary.

Chapter 8

Virtualization-Enhanced TPMs

In this chapter, we present the design of a TPM that supports hardware-based virtualization techniques. Our approach enables multiple virtual machines to use the complete power of a hardware TPM by providing for every virtual machine (VM) the illusion that it has its own hardware TPM. The approach presented in this chapter uses recent developments in the virtualization technology of processor architectures. This chapter shares some material with *Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques* [157].

8.1 Introduction

We have seen in the preceding chapters, that using the attestation mechanisms of the TPM requires a system that is based on virtualization [123]. This is especially true if a system wants to demonstrate to a remote party that it is in a trusted state by using the attestation facilities of a TPM.

Virtualization technologies are getting increasing interest from both industry and academia because they offer an alternative means of hardening an operating system and, thus, of increasing system security [14, 118]. This trend is also reflected in the current hardware architectures. Intel [37, 38] and AMD [6] have added virtualization support to their processor architectures in order to accommodate emerging developments that require the support of multiple virtual machines (VMs) on a single entity. This is not only of interest in the server market, where the execution of multiple commodity operating systems should be supported on a single machine, but also in the client area, where there is a need to completely increase system security and reliability [167].

Unfortunately, as already mentioned in Chapter 7, the TPM was never designed to be used in virtual environments, and is, thus, not capable of being used in a system that is based on virtualization. We circumvented this fact by introducing a software TPM in Chapter 7 that enables the different virtual machines to use their own virtual TPM. However, this approach does not provide the same security level, since the virtual TPM is completely implemented in software and, thus, cryptographic secrets are temporarily not protected by hardware. If for example an attacker gained control of the Management

VM, he might be able to access the secure storage of a virtual TPM and access sensitive VM data.

To provide life-time protection of cryptographic secrets and the possibility of using the functionalities of a hardware-based TPM inside the VMs, we propose to use a TPM that is capable of supporting virtualization with hardware measures. We present the design of such a TPM and show how the interactions between the VMs and the TPM could be realized. The approach presented in this chapter utilizes hardware-based virtualization techniques as provided through Intel's VT processor architecture.

The remainder of this chapter is organized as follows: we first look in Section 8.8 at other work that is related to our proposal. We then extract requirements for a hardware-based virtual TPM and present the main idea of our approach in Section 8.2. In Section 8.3, we explain our architecture and the components that are involved in our approach. Section 8.4 shows how we enable the different VMs to directly use the underlying hardware TPM. Section 8.5 presents the protocol for securely migrating a TPM context to another VM. In Section 8.6, we explain how the endorsement credentials for the different TPM contexts are handled and how the TPM management commands are realized. We summarize with Section 8.9.

8.2 Virtualizing the TPM

One design criteria of currently available TPMs was that they should be easily attachable to a mainboard of a PC. Attaching a TPM to an Low Pin Count (LPC) bus seemed to be a very convenient solution; however, this approach is very problematic as it does not enable the establishment of trust relationships using the TPM in a system that is based on virtualization [156]. One solution for this problem is to use software TPMs [20, 156] that only use the underlying hardware TPM for certain operations. This approach allows the virtualization of the TPM and, thus, the establishment of trust in a system that is based on virtualization. We used this approach in Chapter 7. However, a software TPM cannot provide the same protection level if it is evaluated according to Common Criteria, because software TPMs does not provide mechanisms for preventing unauthorized access to protected data, such as active shields or active security sensors. However, a high evaluation may be necessary if VMs are using a TPM and rely on Trusted Computing technology. To enable a VM to use the full functionality of a hardware TPM without accepting security restrictions, we propose a *multi-context TPM* that is completely realized in hardware.

8.2.1 Requirements

In this section we present the design goals for using a TPM in virtual environments. We have already presented them in Section 7.5.3 and shown in Section 7.7.2 that a software TPM cannot achieve all requirements. We will again recapitulate these design goals since not all goals were achieved by the software TPM.

R.1: Performance This goal states that the overhead from using a TPM in a VM should be negligible compared to directly executing commands on the TPM. The VM should therefore be able to execute TPM commands at nearly the same speed as when these commands are used on a non-virtualized machine.

R.2: Compatibility It should be possible to execute TPM command code in a VM without modifying the code or adapting it to the virtualized environment. Therefore, we explicitly forbid using paravirtualization techniques [185].

R.3: Simplicity A fault in a VMM can cause a failure in all VMs which could result in a crashing VM. A VMM that provides abstraction and sharing of a TPM should therefore be as simple as possible [132].

R.4: Security A TPM that is used in a VM should provide the same security properties as if the TPM would be accessed natively. This goal was not achieved by the software TPM.

R.5: Minimal modifications to the specification If the specifications by the TCG must be modified, these modifications should be as slight as possible, leaving the modifications and the specification as compatible as possible.

8.2.2 Our Approach

The main challenges to providing hardware enhanced virtualization of TPMs is determining how to handle TPM data that is specific for a certain platform, e.g., data that is specific for the physical machine. Such data includes the owner-password of the TPM, the PCRs, the Storage-Root-Key (*SRK*) and the *EK*. This data cannot be shared across all VMs, because if it were, a VM could modify this data, which would result in a TPM state change and, thus, influence the TPM state of the other VMs.

It is necessary to provide every VM with its own instance of a full-fledged hardware TPM, including its own owner-password, PCRs, *SRK*s and *EK*s. Since it should be possible to use the hardware TPM for a theoretical unlimited number of VMs, the multi-context TPM should be flexible and not associate one specific TPM, non-volatile memory region for every VM.

We propose that the multi-context TPM operates on a Control Structure that is loaded into the TPM each time a particular VM operates on its TPM. A VMM is responsible for providing an abstract interface to the underlying hardware TPM and for isolating the different TPM instances. It also performs scheduling operations and is responsible for assigning an unique ID to every VM. This unique ID refers to a specific TPM context. If a VM wants to issue a command to the hardware TPM, the VMM loads the corresponding TPM Control Structure (TPMCS) into the TPM (if not already loaded) and the TPM then operates on this structure. The TPM Control Structure is a data structure that encapsulates all the information needed to capture

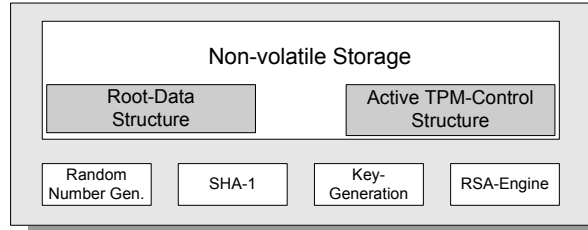


Figure 8.1: Layout of a multi-context TPM

the state of a TPM or to resume a TPM. This approach enables the direct execution of the TPM instructions from a VM on a TPM, and is, thus, very efficient. To provide a hardware protection mechanism, we introduce another privilege level to the TPM. A virtual machine runs in a lower TPM privilege level and, thus, can only operate on its own TPM Control Structure. Operations on other TPMCS can only be done by management commands issued by a VMM. The issuing of such a command by the VM must be intercepted and results in a controlled context switch to the VMM. We will discuss this mechanism in detail in Section 8.4.

8.3 TPM Architecture

The layout of our multi-context TPM is shown in Figure 8.1. Despite the components of a generic TPM, it also provides a second non-volatile storage in which the active TPM Control Structure is loaded. The data of the TPM Control Structure can only be loaded and unloaded into the TPM by TPM management commands. When a TPM Control Structure is unloaded and written back to background storage, it is always protected by secrets that are stored in the root-data structure. Both the root-data structure and the TPM Control Structure hold TPM specific data that is expected to never leave the TPM, as specified by the TCG. This includes Endorsement keys (*EKs*), Attestation Identity Keys (*AIKs*), etc. Only TPM commands that are issued by a VMM can operate on the root-data structure. Replay attacks on the TPM Control Structure are, in principle, possible. However, these attacks are also particularly problematic in the current specification. To overcome these problems, an extension of the TPM specification as proposed by us in [91] could be used. Note that such replay attacks could basically be used to replay old PCR values stored in the TPM Control Structure. However, these attacks would require a modified and malicious VMM and would thus be detected since the hashvalue of the integrity of the VMM is stored in the root-data structure of the TPM (see Section 8.3.3).

We assume that our multi-context TPM is either integrated on the CPU or on a fast bus, and has a direct connection to the CPU, leaving it essentially free of hard speed constraints compared to the LPC bus.

8.3.1 TPM Protection Rings

To provide the TPM with a hardware-based protection architecture and to isolate one VM TPM context from another, we introduce hierarchical protection domains (protection rings) into the TPM. These protection rings are shown in Figure 8.2 and distinguishes two different TPM modes of execution. For this purpose, we introduce a 1-bit non-volatile control register (CR), which the actual TPM state refers to. Every time a context switch occurs, i.e., a controlled state transition from non-privilege TPM mode to the privilege TPM mode and vice versa, the TPM control register is set appropriately. The Figure also shows which x86 CPU modes, and which software (VMM or guest-OS), is executed in which ring. If our TPM is directly integrated inside the CPU, and is therefore able to use the protection rings of the CPU, the integration of the CR inside the TPM could be omitted. Nevertheless, our proposed architecture requires a direct interaction with the CPU in order to satisfy the necessary protection domains and to ensure that context switches are reliably enforced. Hence, we require a strong collaboration between CPU and TPM as we will see in the following sections.

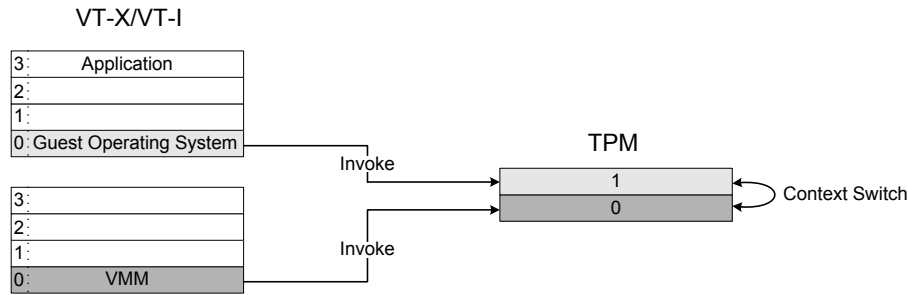


Figure 8.2: Privilege level of a multi-context TPM

The VMM ought to be run in the privilege VMX root-operation of the CPU as well as in the TPM privilege mode. The VMM runs on a higher privilege level of the TPM and is, thus, able to manage TPM state transitions. However, under the assumption that the authorization data (e.g. owner password) of the TPM are kept secure inside the VM, the VMM cannot inspect the communication between VM and TPM. Parts of the communication between TPM and the software stack are encrypted with a session-key that is derived from the authorization data (Cf. [171], pp. 60).

8.3.2 TPM Back-End Device Driver

The TPM back-end device driver is integrated into the VMM and therefore runs in ring 0 of the CPU's VMX-root mode. Its main purposes are isolating the different TPM interfaces and scheduling the TPM commands invoked by the VM. It also maintains a data set including the unique IDs of each TPM context as well as their associations to the VMs. Note that this TPM back-end device driver does not need to implement the

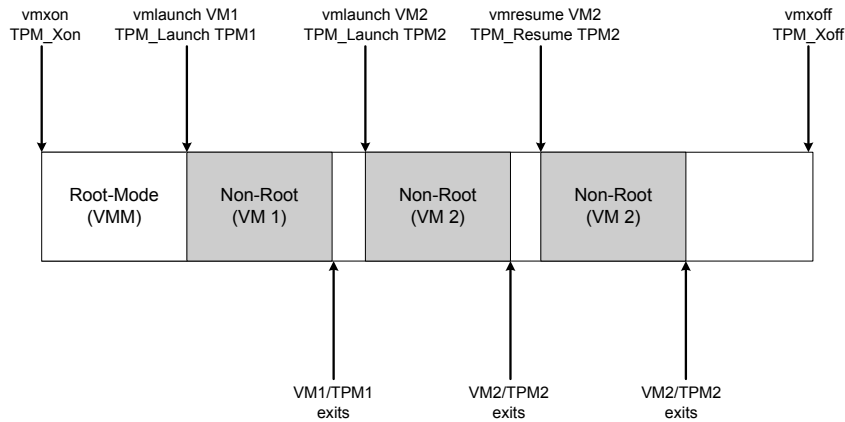


Figure 8.3: Transition between different TPM contexts

full software stack of the TPM, since conventional operations on the TPM are issued by VMs that implement their own software stacks.

Figure 8.3 shows a typical sequence of operations inside a virtualized TPM environment. The white areas of the Figure show the operations of a VMM that set up the environment for every VM and its corresponding TPM context. The gray areas represent two different VMs with their corresponding TPM contexts. The transition between VMM and VM are invoked by special instructions, which can only be executed in the TPM privilege mode.

Every TPM that has been turned on in normal mode can be made to enter the virtualization mode by executing the `TPM_Xon` operation. The transition of the TPM contexts needs to be synchronized with the transitions between the different modes of execution on the Intel VT-X/I architecture, in order to support direct native execution on the TPM. This property is discussed in detail in Section 8.4.

The VMM assigns for every TPM context, as well as for every VM instance, an execution time in which the VM is allowed to execute operations on the TPM. This execution time not only includes the time a VM is allowed to use the TPM, but also how long the VM is allowed to use the CPU. Determining the exact execution time is task of a scheduling algorithm, such as round-robin. Thus, all VM does not necessarily receive the same execution time to execute instructions on the underlying resources. If this execution time has passed, the VMM regains control of the TPM and assigns the next context of the TPM. To keep the VMM from losing control of the TPM, the VMM must set an *interval timer*, that states how long the VM is allowed to use the TPM and the underlying processor. This interval timer is set by the VMM before passing control to a VM. It counts down clock cycles and if it reaches zero, triggers an interrupt. The interrupt then passes the control back to the VMM.

8.3.3 TPM Control Structure

A transition from one TPM context to another is controlled by the VMM through a TPM Control Structure (TPMCS). The VMM associates to every TPM context its own TPM Control Structure. This structure holds all state specific TPM data as shown in Figure 8.4 and only the TPM can operate on this structure. Each time the VMM closes a TPM context and spawns a new context, the old Control Structure is saved on the background storage and the new Control Structure is loaded into the TPM. Since this Control Structure holds sensitive TPM data, cryptographic measures must be in place to prevent unauthorized modifying of a Control Structure.

To protect this TPM structure from unauthorized modifications, we propose using the Storage Root Key of the TPM to seal this structure to the current platform configuration of the system. This *SRK* is stored inside the TPM root-structure and is only accessible by the TPM in the root-mode of the TPM. Sealing the structure to a set of platform configuration registers is necessary for ensuring that the VMM is in its initial state and trusted. In contrast to the *SRK* of the VM, the *SRK* of the root-mode will never leave the TPM.

Fields of the TPM Control Structure
Storage Root Key
PCRs [16..23]
Attestation Identity Keys
Endorsement Key
Endorsement Credential
Monotonic Counter Values
Values of the non-volatile storage
Delegation Tables
TPM context data
DAA TPM specific secret (f)
TPM_PERMANENT_FLAGS/DATA
TPM_STANY_FLAGS/DATA
TPM_STCLEAR_FLAGS/DATA
Authorization data

Figure 8.4: TPM Control Structure

The TPM also possesses a number of special purpose registers, such as the tick counter or the PCRs. Tick counter and the lower values of the PCRs are special, since both should be consistent in all VMs. We suggest that the tick counter and the PCRs [0..15], which hold the data from the bootstrap procedure and the VMM's integrity, be stored in a special region of the TPM. Every TPM context should be able to read the values stored in these registers. Writing to this registers is only possible if the TPM is in the privilege mode and its CR is set to 0. To enable a VM to attest its configuration, the TPM computes the signature on PCRs [0..15] and PCRs [16..23] using one *AIK* stored inside the loaded TPM Control Structure.

8.3.4 Extended Instruction Set

To realize the management features of the TPM, we extend the specification to a number of additional TPM commands. These commands manage the TPM state and are used to control the different TPM contexts. All commands are executable only by the VMM, since they must be executed in ring 0 of the TPM. If one such command is executed in ring 1 of the TPM, they trap into the VMM, where a dispatcher emulates the instruction. In the remainder of this paper, these instructions are referred to as sensitive instructions. The TPM also supports a number of instructions that allow a TPM context of being migrated to another TPM. We will explain these instructions in detail in the remainder of this work.

- `TPM_Xon`, `TPM_Xoff` enables/disables the second privilege level of the TPM
- `TPM_Launch` creates and launches a new TPM context and an empty TPMCS
- `TPM_Resume` loads an existing TPMCS into the TPM and launches the corresponding TPM contexts
- `TPM_Exit` saves an existing TPMCS and seals it to certain PCRs
- `TPM_Clear` deletes an existing TPMCS and the corresponding TPM context
- `TPM_Migrate` migrates an existing TPMCS from a source TPM to a destination TPM
- `TPM_InitializeMigration` initializes the migration procedure on the source TPM
- `TPM_InitializeImport` initializes the migration procedure on the destination TPM
- `TPM_Import` imports a migrated TPMCS into the destination TPM

These instructions also control which TPM context and which corresponding VM will receive the underlying hardware TPM, thus, which VM is allowed to operate on the TPM.

8.4 Direct Native Execution

For virtualizing a resource, there exist a number of different techniques. A processor is typically *shared* by a number of processes or VMs and every process is allowed to use the resource for a particular time period. This technique is also adapted to our approach. However, other techniques also exist, for example, *partitioning* which is normally done for background storage (disks); and *emulation*, which is for example used in the software TPM approach [20] or if a processor is completely emulated [17].

The main advantage of the sharing technique is that it enables *direct native execution* and the use of efficient virtualization, as stated by Popek and Goldberg's first Theorem

[69]. To enable an efficient virtualizable TPM, we adapt the TPM so that it is possible to execute nearly all instructions directly on the TPM and, thus, to satisfy Popek and Goldberg’s first Theorem.

In order to achieve direct native execution, it is absolutely necessary that sensitive instructions trap into the VMM. Sensitive instructions are instructions which can result in an illegal TPM state change. If such a sensitive instruction is executed by a VM it must trap into the VMM where a dispatcher routine emulates the instruction. However, since the TPM does not possess its own program counter, it cannot autonomously trap into the VMM if it encounters a sensitive instruction. This property is similar to sensitive x86 processor instructions such as `POPF`, which do not perform a trap if they are executed in the non-privilege processor mode [129].

Since TPM state transitions and VM state transitions are synchronized and controlled by the VMM, the VM, which is actually the owner of the processor also occupies the TPM. Therefore, only the VM that currently has execution time on the processor can issue commands to the TPM. Consequently, we only have to consider the case in which a sensitive instruction is executed in ring 1, such as `TPM_Exit`.

8.4.1 Handling Sensitive Instructions

If a sensitive instruction, such as `TPM_Exit`, is executed in ring 1, the TPM must trap into the VMM. Sensitive commands must be executable only by the VMM, otherwise, a VM could overwrite values of a TPM Control Structure which belongs to another TPM context. Since a VMM cannot inspect all commands sent by a VM, the TPM must decide whether or not this command is to be executed. If the TPM receives a management command, it should check internally whether the TPM CR is in ring 0. If not, the TPM should jump to the dispatcher routine running inside the VMM. Note that either the VMM or the VM could issue a management command while the TPM’s privilege level is set to 1. This is due to the fact that the status register of the CPU and the TPM are not necessarily synchronized. Thus, it could happen, that the CPU is operating in VMX-root mode, while the CR of the TPM is still set to 1. In that case, the CR of the TPM must first be set to 0 before the TPM can execute a VMM issued management command.

Since the TPM cannot directly modify the program counter of the CPU if it receives a privilege command, an exception must be caused that allows the CPU to jump into the correct dispatcher routine inside the VMM. We propose to use *interrupt requests* as they are typically used by other PCI devices through the `INTR` bus signal [39]. If the TPM is integrated inside the CPU, the TPM might be able to directly modify the program counter and, thus, jump directly into the dispatcher routine of the VMM.

Algorithm 4 shows the steps performed by the CPU and the TPM respectively. If an exception happens, the TPM signals an interrupt by emitting the `INTR` bus signal. The processor then reads from the system bus the interrupt vector number provided by an external interrupt controller. This vector number signals to the CPU which interrupt is caused. Based on the information given inside the VMCS held by the VT-I/X CPU, the CPU performs a `vmexit` and jumps into the corresponding dispatcher routine of

Algorithm 4: Handling sensitive instructions

```

TPM receives TPM_Exit ;
if CR0 = 0 then
|   Execute TPM_Exit;
else
|   CR0 := 0 ;
|   Execute INTR ;
|   Interrupt causes vmexit ;
|   CPU loads exit information from VMCS ;
|   CPU sets PC to VMM entry point address ;
|   VMM executes/emulates TPM.Exit ;
|   CR0 := 1 ;
|   vmentry (VMCS) ;

```

the VMM. For this purpose, the VMCS of the VT-I/X architecture must hold the information where the TPM dispatcher routine is located in memory.

8.4.2 Scheduling the TPM

The VMM is responsible for the transitions between different VM contexts. This approach is analogous to scheduling VMs. However, the problem is that if the TPM is not directly integrated into the CPU, the privilege level of the TPM does not directly depend on the privilege level of the CPU. Thus, a *vmexit* does not directly set the control register of the TPM to zero. Therefore, a controlled state transition must be passed to the TPM, which then issues an interrupt that jumps to a concrete entry point of the VMM.

Figure 8.5 shows the actions performed by the VMM in retiring one VM and activating the next VM. The figure also shows the value of the TPM control register. The CPU performs a *vmexit* that is caused by the interval timer and sets the PC to a specific entry point of the VMM. This controlled state transition also stores the actual state of the VM inside the VMCS. The VMM then determines which VM is next in line to use the processor. The VMM then closes the TPM context by sending a *TPM.Exit* command to the TPM. The TPM resets its control register to zero, issues an interrupt and delivers the corresponding interrupt vector number using an interrupt controller to the CPU [39]. Based on the actual interrupt descriptor table (IDT), the CPU again jumps to a specific entry point of the VMM, where the instruction is again sent to the TPM. The VMM then resumes the next TPM context by sending the *TPM.Launch* command together with the stored and encrypted TPMCS to the TPM. Afterwards, the VMM resets the interrupt timer and relinquishes control to the VM by loading the VM state information into the CPU (*vmentry*).

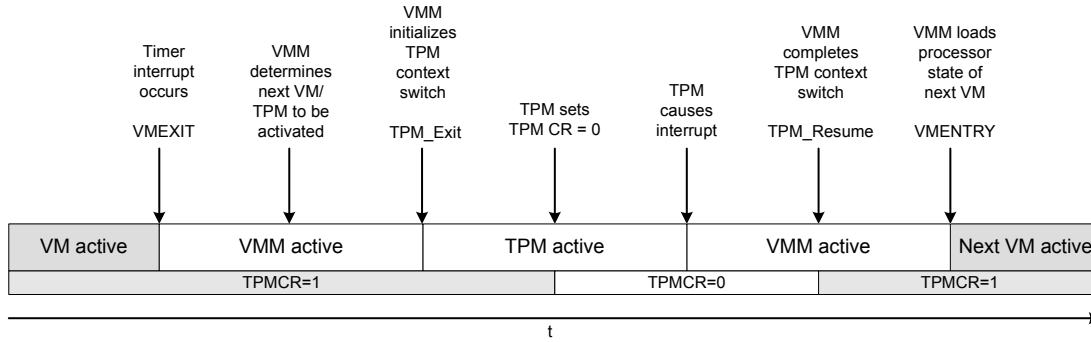


Figure 8.5: Workflow of retiring one VM/TPM and activating the next VM/TPM

8.5 Secure TPM Context Migration

VMs can be migrated to other platforms. Since a VM might have stored cryptographic keys inside our multi-context TPM, we have to consider how a complete TPM context is securely migrated to another TPM. Basically, all state information of a TPM context is stored inside the TPM Control Structure. However, since this Control Structure is bound to a specific TPM, it cannot be transferred directly.

To prevent a TPMCS's being migrated to multiple contexts or an old TPMCS's being again replayed into the system, we propose the use of the monotonic counter of the TPM to synchronize all existing TPM contexts with the root-structure of the TPM. For this purpose, each TPMCS and the root-TPM structure hold a register in which both are incremented each time a TPMCS is migrated. We refer to this register as the *migration counter*. Before a TPMCS can be migrated, the TPM internally verifies whether both migration counters have the same value. If an already migrated TPMCS is loaded again in the TPM, its migration counter differs from the one stored inside the TPM and the TPM will refuse to operate on this structure. This check must always be performed before a TPM context is allowed to operate on a TPMCS. Note that the migration counter cannot be reset and can only be incremented. Before a TPMCS can be migrated to another context, it must be ensured that the destination platform is in a trusted state. This should be done by remotely verifying the platform state of the destination platform using platform attestation. For this purpose, a secure attestation channel, such as the one established by Protocol 3.6.3 must be established. We assume that a secure attestation channel has been established between both entities and that our migration protocol runs inside the resulting channel.

Figure 8.6 shows our migration protocol. It has similarities to the concept the TCG introduced with migratable keys and simply transfers an encrypted TPMCS blob to the other TPM. The TPMCS is directly bound to random nonces generated on the source and the destination platform to prevent a TPMCS from being migrated to multiple TPMs.

In the first two steps, the TPM generates a non-migratable TPM key K_A and certifies this key to provide assurances that it is held in a protected storage of a genuine

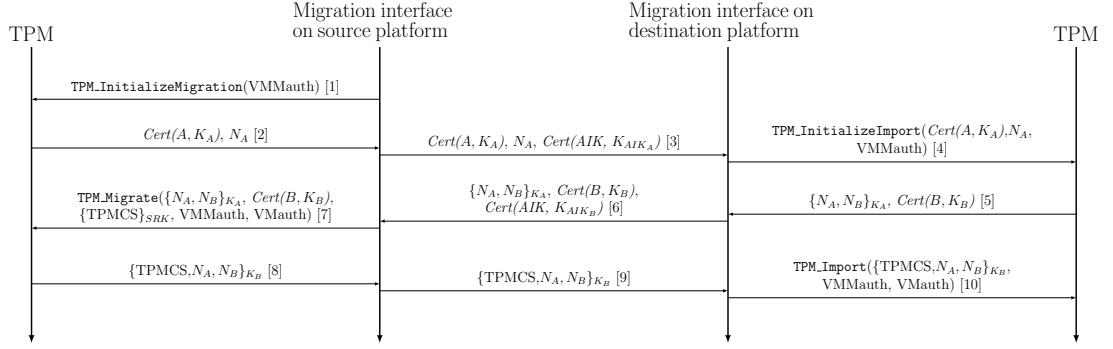


Figure 8.6: High-Level Migration Protocol

TPM. This key is then transferred together with a nonce (N_A) and the AIK certificate to the destination platform, where the import is initialized. The migration interface verifies that the query is coming from a genuine TPM and that the K_A is protected by a TPM, using the provided certificates (Cf. [174], pp. 130). The destination TPM initializes the import procedure by generating and certifying its own non-migratable TPM key (K_B). For this purpose, the interface of the migration platform must provide owner authorization, including N_A and the certificate of the source platform. Then, the TPM delivers the encrypted package consisting of N_A and N_B to the migration interface. The migration interface collects all certificates that are necessary to prove that K_B is a TPM protected key and transfers the package back to the source TPM. The source migration interface then provides owner authorization to the TPM and delivers the received encrypted package, including the TPMCS that is to be migrated to the TPM. The TPM verifies authorization of the command and checks internally whether the migration counter conforms to the migration counter of the TPMCS that is to be migrated. If everything checks out, the TPM increments the internal migration counter and the migration counter of all other TPMCS that reside on this system. The TPM must ensure that the migration counter of each TPMCS is only incremented once, since otherwise, an old TPMCS could be replayed into the system.

The TPM then encrypts the TPMCS and the nonces with the delivered TPM key of the destination platform and transfers the package to the destination platform. The TPM of the destination platform imports the package and activates the TPMCS if the nonces and the asserted authorization data are correct. The migration procedure finishes by setting the migration counter of the TPMCS to the destination's TPM actual value of the internal migration counter and by re-certifying the EK of the TPMCS.

8.6 TPM Credentials

Every TPM is uniquely identifiable by the Endorsement Key (EK). This EK is pre-installed by the manufacturer of the TPM and is used to generate the owner-password.

In order to provide full-fledged TPMs for every TPM context, every context needs to possess its own *EK*. However, it should also be possible for a verifier to decide whether the *EK* of a TPM context is an authentic *EK*, i.e., generated and protected by a hardware TPM. It is possible for a manufacturer to generate a number of *EK*s and integrate them on our multi-context TPM. However, we believe that this process is not feasible, since this would require an additional overhead for the manufacturer. In addition, all *EK*s and certificates could be migrated out of the domain of the multi-context TPM and deplete the device of its pre-installed *EK*s.

1.	VMM \rightarrow	TPM	: TPM_OSAP(<i>authData</i>)
2.	VMM \leftarrow	TPM	: <i>H</i>
3.	VMM \rightarrow	TPM	: TPM_Exit(<i>H</i>)
4.		TPM	: Store TPMCS values in D_{n+1} : Create PCRInfo Structure : $\{D_{n+1}\}_{SRK} := \text{Seal}(D_{n+1}, \text{PCRInfo}, H)$: $\text{sig}_{n+1} := D(D_{n+1}, K_{TPM}^{-1})$
5.	VMM \leftarrow	TPM	: $\{D_{n+1}\}_{SRK}, \text{sig}_{n+1}$
6.	VMM \rightarrow	TPM	: TPM_Resume(<i>H</i> , $\{D_n\}_{SRK}, \text{sig}_n, \text{authData}$)
7.		TPM	: Check if <i>migration counter</i> $\stackrel{?}{=} \text{TPMCS.migration counter}$
8.			: $D_n := \text{Unseal}(\{D_n\}_{SRK}, H)$: Verify sig_n with D_n : Load D_n into TPMCS : CR0 := 1
9.	VMM \leftarrow	TPM	: TPM_SUCCESS

Figure 8.7: Exiting and Resuming a TPM context (CR of the TPM is set to 0)

We propose to establish a certificate chain with the *EK*, which is held in the root-TPM structure as a root node. For every TPM context, the TPM generates its own *EK*, which then becomes certified by the root *EK*. Generating *EK*s directly on the TPM is a feature that current TPMs already support (Cf. [175], pp. 143). This process allows us to obtain AIKs for every TPM context simply by verifying the certificate chain of the *EK*. If a TPM is migrated, the *EK* of the migrated structure must then be re-certified with the *EK* that resides on the destination platform.

In the following, we provide the protocols for creating and exiting a TPM context. These protocols are exemplary for the other TPM commands that we introduced in 8.3.4.

8.6.1 Spawning a TPM Context

The protocol for creating a new TPM context is shown in Figure 8.8. First, a TPM Object-Specific Authorization Protocol (OSAP) session is created between the TPM and the VMM. This command requires authorization (owner-password and *SRK*-password) and computes a cryptographic secret and a handle (*H*) to this session; both to protect

the whole session traffic (Cf. [171], pp. 71). The returned handle is then used to issue the `TPM.Launch` command. The TPM then internally creates a new TPM context and adds an *EK* and the corresponding credential to this structure.

1.	VMM →	TPM	:	TPM.OSAP(<i>authData</i>)
2.	VMM ←	TPM	:	<i>H</i>
3.	VMM →	TPM	:	TPM.Launch(<i>H</i>)
4.		TPM	:	Create empty <i>D</i>
5.		TPM	:	Create 2048-bit Endorsement key-pair (K_{vEK}, K_{vEK}^{-1})
			:	Sign K_{vEK} with K_{EK}^{-1}
			:	Add Cert(K_{vEK}) and K_{vEK} to <i>D</i>
6.		TPM	:	Load <i>D</i> into TPMCS data field
7.		TPM	:	CR0 := 1
7.	VMM ←	TPM	:	TPM.SUCCESS
8.	VMM		:	LaunchandMeasureVM

Figure 8.8: Creating a new TPM context (CR of the TPM is set to 0)

After the TPMCS has been created, the VMs integrity is measured [156] and the obtained measurements are added to the PCR value of the current loaded TPMCS.

8.6.2 Exiting a TPM Context

Exiting requires that the current loaded TPMCS is stored on the background storage. To prevent an attacker from being able to inject corrupt data structures, a TPMCS is sealed and signed with keys that are protected by the TPM. Figure 8.7 shows how this is realized. The TPM creates a special purpose data structure D_n in which the current values of a TPMCS are stored. This data structure is then sealed to the actual platform configuration, which must also include the integrity of the VMM.

This can easily be achieved by using the Safer Mode Extension (SMX) of the Intel TXT technology [38] or the AMD Secure Virtual Machine technology [6]. Both provide a special CPU command; in the case of Intel, it is called *getsec* and in the case of AMD, it is called *skinit*. These commands measure the VMM into a PCR and create a so-called *Dynamic Root of Trust for Measurement* (DRTM).

We denote the sealing of the structure D_{n+1} at a specific time Δt with $Seal(D_{n+1}, \text{PCRInfo}, H)$. H is a key-handle for the storage root key and PCRInfo is a `TPM_PCR_INFO` structure that contains the information to which PCRs D_{n+1} will be bound. The operation to unseal is denoted as $Unseal(\{D_{n+1}\}_{SRK})$ where for simplicity reasons $\{D_{n+1}\}_{SRK}$ also includes the structure of the platform configuration registers. Unsealing the $\{D_{n+1}\}_{SRK}$ at $\Delta t + x$ is then only possible if the current platform configuration is equal to the platform configuration that $\{D_{n+1}\}_{SRK}$ is bound to.

The data structure (D_n) is additionally signed using a TPM specific signing key (K_{TPM}^{-1}), which is stored in the root-structure of the TPM only for that purpose. The sealed data structure and the corresponding signature is then passed to the VMM, which stores it locally. The VMM then executes the `TPM.Exit` command and delivers

the sealed data structure and the corresponding signature of the TPM context which should be resumed by the TPM. The TPM internally verifies whether the *migration counter* of the TPMCS is consistent with the one stored inside the *migration counter* of the root-data structure. This verification prevents against an already migrated TPM context or a replayed TPM context being again loaded into the TPM. The TPM then unseals the structure and verifies that the signature sig_{n+1} is a valid signature of D_{n+1} . If the signature is valid, implying that the data structure was generated on this specific TPM, the TPM loads the values D_{n+1} into its internal memory.

8.7 Satisfied Security Requirements

In contrast to our first approach that used a software TPM to virtualize a TPM (compare 7.5.3), we are now directly using a virtualization-enhanced hardware TPM. This hardware TPM satisfies all requirements (R.1 - R.3) that a software TPM satisfies. In addition, it also satisfies the security requirement (R.4) since all cryptographic secrets are protected by hardware measures and can thus only be accessed by hardware.

8.8 Related Work

Another very related proposal has been made by England and Loeser [57]. The authors propose to safely share a TPM among other virtual machines by paravirtualizing the TPM. All TPM data that cannot be shared is paravirtualized, i.e., the state of each VM's TPM is stored in software. This is for example true for the PCRs which cannot be shared amongst all VMs. However, the approach has two major drawbacks. First, it does not provide the same security level as our proposed multi-context hardware TPM, and, Second, the *EK* must be shared through all VMs which is a potential security vulnerability.

Sadeghi et al. propose in [136] a virtual TPM architecture that supports property-based attestation. Their approach extends a virtual TPM with mechanisms that allows migrating a vTPM instance to another platform that offers the same platform properties. However, in contrast to our work, the approach by Sadeghi et al. is also based on the software TPM [20] and, thus, does not provide lifetime protection of secrets.

Similar to a TPM, ARM has introduced the TrustZone technology [5] which is an extension to the ARM architecture. The TrustZone technology tries to ensure reliable implementation of security critical applications by adding a second secure processor execution mode in which applications with a higher protection level are run. In contrast to the TPM, ARM TrustZone technology is implemented within the microprocessor core itself. However, TrustZone does not provide the flexibility of a TPM and does only support a small set of the functionalities that a TPM supports.

Intel and AMD have recently introduced the Trusted Execution Technology (TXT) [38, 180] and the Secure Virtual Machine technology [6], respectively. Both architectures extend the processor instruction set with a number of additional special purpose instructions, that directly communicate with a TPM. In addition, these architectures

implement virtualization technology and are, thus, capable of supporting different efficient VMs with hardware measures.

8.9 Summary

Trusted Computing technologies provide a sound way of securing computer systems and also a technological means for trust establishment. For this purpose, the Trusted Computing Group introduced a hardware module called trusted platform module (TPM) that protects cryptographic secrets and is capable of acting as a trust anchor. However, the TPM cannot be used directly in next-generation operating systems that utilize virtualization technologies. In this chapter, we proposed an efficient approach for using TC-technology in virtual environments. Our approach extends the TPM specification and shows how a hardware TPM that is capable of supporting virtualization with hardware measures should be designed. To provide hardware-based protection domains, we introduced a second TPM privilege level and a TPM Control Structure. The combination of both concepts allows a virtual environment to directly operate on the TPM without loss of security properties. Since the approach we presented in this chapter utilizes recent developments in the virtualization technology of processor architectures, it could easily be adapted to integrate Trusted Computing technology in next-generation processor architectures. In that case, highly-efficient and secure Trusted Computing technology would be available to next-generation operating systems that are based on virtualization.

Part IV

Transaction Software

Chapter 9

Secure Transaction Software

This chapter presents a secure transaction software that can be used for sensitive e-commerce transactions. This transaction software runs on our security architecture presented in the preceding chapter and is, thus, capable of providing a proof that it is in a trusted state. Parts of this chapter have been previously published in *An Approach to a Trustworthy System Architecture Using Virtualization* [156] and *Towards Secure E-Commerce Based on Virtualization and Attestation Techniques* [158].

9.1 Introduction

In the preceding sections, we presented the design and implementation of an attestation-supporting security architecture. However, a security architecture cannot provide protection for confidential data if the application that is used as input interface and transaction software suffers from a wide range of vulnerabilities. This shows that besides an underlying security architecture, the need for a secure transaction software evolves.

In today's personal computers, security sensitive transactions are often facilitated by the use of web-browsers such as Microsoft's Internet Explorer, Mozilla's Firefox or Apple's Safari. However, such browsers were never explicitly designed for acting as a secure transaction software. Further complicating this problem is the issue of browser help objects (such as flash players and video codecs) which extend the complexity of these web-browsers. Since these components could be used to inject malicious code and users are typically not adept at separating useful applications from malicious ones, these components should not be part of a secure transaction client. Mozilla's Firefox, the most widespread web-browser, as an example, consists of around 1 million lines of code, showing that the code complexity is huge. Because of their huge complexity and the ability to execute additional *programs*, such as scripts, active code or applets, web-browsers have become a de facto operating system [40]. However, since many software engineering studies have shown that the code complexity is correlated with the number of errors and defects [151, 15], a secure transaction software should be as simple as possible.

To make matters worse, the unsecure web-browsers typically run on unsecure and unreliable operating systems. As shown in the preceding chapter, one vulnerability of the underlying operating system can be exploited, thereby allowing an attacker to surreptitiously install subversive programs, such as malware and spyware, which can eavesdrop, record and distribute a user's actions, passwords, credit card information, bids in auctions or other sensitive data. As a result, it is difficult for users to ascertain whether or not their computer's software system can be trusted. However, such assurances are necessary if users are to become more comfortable purchasing goods on-line [59].

Another aspect that needs attention is that even if we are to assume trusted user environments, a system may be vulnerable to man-in-the-middle (MitM) attacks. MitM attacks pose a serious threat to current electronic commerce applications that are based on TLS [112]. These attacks exploit the fact that ordinary users often improperly verify a merchant's certificate. Even if the users carefully examine the certificate presented by the merchant, they cannot be certain that the merchant's identity is authentic, since their own software could have maliciously modified the certificate presented to them. In order for users to place trust in their system for use in e-commerce, it is, therefore, necessary to provide them with assurances that:

1. they have a trusted client configuration.
2. authentication data, such as passwords, cannot be accidentally transferred to an improperly verified server.

In this chapter, we present a secure transaction software that runs in the virtualization-based environment presented in Chapter 7 to achieve strong isolation between compartments of different trust levels. This setup ensures that the sensitive e-commerce transaction client is immune from infection by malicious processes running in different compartments. To prevent MitM attacks on TLS and to ensure that the client application is trusted, we propose several security protocols that are based on our proposed attestation protocols. Using these protocols, we can ensure

1. that the client configuration remains untampered and trusted for the duration of the transaction.
2. that confidential data, such as authentication passwords, are only accessible to the electronic commerce server to which the users intend to transfer their data.

This chapter is organized as follows: Section 9.2 describes our e-commerce architecture; in Sections 9.3 and 9.4, we discuss how to ensure that a client's platform is trusted and how to establish a secure channel with a remote host that is resistant to MitM attacks. In Section 9.5, we provide an informal security analysis of our proposal. In Section 9.6, we discuss some of the implementation issues that arise. In Section 9.7 we look at other work that is related to this chapter; and finally, we summarize with Section 9.8.

9.2 Overall E-Commerce Architecture

In contrast to existing e-commerce architectures, which often only consist of a server and a client, our proposed architecture has an additional component. We employ a trusted third party (TTP) that is responsible for distributing the secure transaction software to clients and for subsequently validating the platform configuration of the client platforms. The TTP maintains a data set of trusted reference values, which are compared to the client's platform configuration to determine the trust level of the client platform. The reason for the TTP is that a typical server does not necessarily have the potential to decide whether or not a particular client platform is trusted. In addition, privacy issues would arise if a server were to know which software versions the client uses.

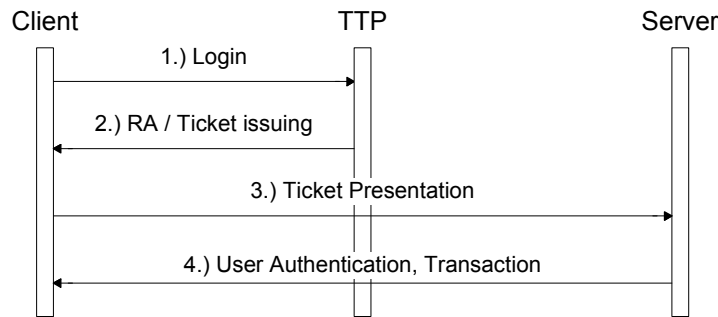


Figure 9.1: Transaction Model

Figure 9.1 shows our transaction model. When the secure transaction software (STS) running on the client connects to the platform provided by the TTP (step 1), the client platform's configuration is verified via platform attestation (step 2). When the client's platform is deemed trusted, the TTP issues a ticket that vouches for the platform configuration of the client's platform. This ticket will later be presented as a credential so as to allow the platform configuration to obtain access to services offered by the server (step 3).

9.2.1 Client Architecture

We use our proposed security architecture from Chapter 7 to enable the establishment of several different execution environments that are strongly isolated from one another. The secure transaction software (STS) runs in one isolated execution environment to ensure that other applications cannot interfere with it. The architecture is depicted in Figure 9.2. The TVM runs the STS and a tiny OS with a minimal set of software components (this reduces the possible number of security vulnerabilities). The OS and the STS are part of a virtual appliance (VA), which is a fully pre-installed and pre-configured virtual machine image. To further reduce vulnerabilities, the TVM is stateless, which means that any modifications in the guest system cannot be written

back to the disk image. The TVM is also equipped with a secondary disk image that can be used to store transferred data and information about former transactions.

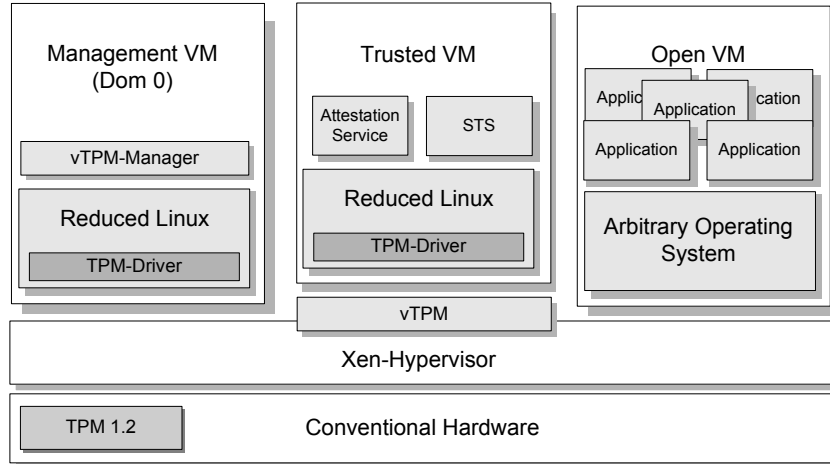


Figure 9.2: The secure transaction software runs in the TVM of our proposed security architecture

9.2.2 Server Architecture

The server also runs the security architecture from Chapter 7 to provide better protection against attacks and to detect possible tampering with the software configuration. In one TVM, the server executes a trading software that is able to communicate with the STS and able to validate tickets issued by the TTP. The trading software is comparable to a conventional shop system that has been modified to support the creation and verification of digital signatures and to verify the authenticity and validity of the ticket that vouches for a trusted client configuration. Since this trading software runs in one TVM, it can also make use of attestation techniques in order to prove that its software configuration is in a trusted state. To satisfy scalability issues and to overcome the performance bottleneck of a TPM, one of our protocols proposed in Chapter 4 can be used.

9.2.3 Ticket-Based Attestation

We use the ticket-based remote attestation scheme as introduced in Chapter 5 to verify platform integrity, thereby validating that a communication partner's platform is trusted. Even though our approach introduced in Chapter 5 can handle multiple validators, each of which verifies the integrity of one software component, we only employ one validator, namely the TTP, which verifies the integrity of the entity's whole platform integrity. The reason for this is that due to the strong isolation between execution environments provided by our security architecture, it is sufficient to only validate the trust level of the TVM and its underlying layers. We assume that the TTP also has

the potential to validate the underlying layers, which only consist of the hypervisor, the Management VM and the Bootloader. Note that there is no need to validate the complete boot-sequence including BIOS and ROM, since the hypervisor can be executed using the *skinit* [6] or *getsec* instruction [38], thus establishing a DRTM [92].

9.3 Trusted Transaction Software

At present, users typically have little indication of the trust level of their platforms including their transaction software. By isolating different compartments, virtualization technologies reduce code complexity and the vulnerability to malicious code, thereby minimizing the risk of a platform becoming compromised. However, even with virtualization, users who interact with their platforms have no means of confirming that their software is trusted. To cope with this problem, attestation techniques could solve this problem by verifying the platform configuration through a remote entity and denying access to a remote service if the platform is not considered trusted. However, adapting this mechanism to e-commerce is challenging since malicious software components could simulate a successful attestation as well as a successful e-commerce transaction whilst eavesdropping confidential data. As a result, the secure attestation protocols as proposed in Chapter 3 cannot directly be applied.

9.3.1 Attestation Protocol

To ensure a trusted client configuration and to prevent malicious client applications from spying on user entered data, we introduce a concept that shares some similarities with the sealing concept, but circumvents the need for coping with legitimate system updates. Legitimate system updates may occur very frequently even if a virtualization-based system architecture, such as the security architecture in Chapter 7 is used. To overcome this challenge, we subdivide the attestation process by adding an additional initialization phase, which is only executed once. We assume that the platform configuration of the client is trusted in this initialization phase. This phase could, for example, be executed directly when the platform is first set up. It should be noted that the initialization phase is only vulnerable to attacks that modify the complete attestation process by simulating a trusted TTP.

We use an adapted version of Protocol 5.4.1 introduced in Chapter 5. This is due to the fact that the platform configuration of our client may change very often. This makes the approach used in Protocol 5.4.1 (i.e., generating a new TPM-bound key each time the platform configuration changes) rather impractical. However, the basic security mechanisms of Protocol 5.4.1 and the protocol proposed herein are identical.

The attestation protocol is divided into two separate protocols. In the *initialization phase* the users choose a secret, which could also be a picture or a fractal structure, to enhance usability. This secret is then cryptographically bound to the users' TPM-enabled platform. If the platform configuration is trusted, the *attestation phase* should

decrypt this secret and display it to the users. This process assures the users that their platform is in a trusted system state.

The complete protocol run of the initialization phase is shown in Protocol 9.3.2. First, a secure attestation channel between B and T using Protocol 3.6.3 is established. B is the entity that runs the TVM and T is a trusted third party. This secure attestation channel is then used to transfer the attestation ticket and bind a user-specific secret to platform B .

Protocol 9.3.2: Initialization Phase

SUMMARY: B answers the attestation challenge of T

RESULT: Establishment of a secure attestation channel: secret generation and binding to a platform

1. *Protocol messages.*

$$T \rightarrow B : Nt, g^t \bmod p, g, p \quad (1)$$

$$T \leftarrow B : \text{Cert}(vAIK, K_{vAIK}), \{Nt, PCR, g^b \bmod p\}_{K_{vAIK}^{-1}} \quad (2)$$

$$T \rightarrow B : \{Nb, g^t \bmod p\}_{K_{TB}} \quad (3)$$

$$T \leftarrow B : \{Nb, Nt, SML, g^b \bmod p\}_{K_{TB}} \quad (4)$$

$$T \rightarrow B : \{\{g^b \bmod p, g, p, \text{timestamp}\}_{K_T^{-1}}\}_{K_{AB}} \quad (5)$$

$$T \leftarrow B : \{s, \text{Cert}(TPM, K_{TPM})\}_{K_{TB}} \quad (6)$$

$$T \rightarrow B : \{\{\{s\}_{K_{TPM}}\}_{K_{TTP}}\}_{K_{TB}} \quad (7)$$

2. *Protocol actions.*

- (a) *Precomputation by TTP.* T selects an appropriate prime p and generator g of \mathbb{Z}_p^* ($2 \leq g \leq p-2$).
- (b) T chooses a random secret t , $2 \leq t \leq p-2$, and computes $g^t \bmod p$.
- (c) *Attestation challenge.* T sends message (1) to B .
- (d) B also chooses a random secret b , $2 \leq b \leq p-2$, and computes $g^b \bmod p$.
- (e) B computes the attestation response message by signing the own public key, Nt and a set of PCRs using an $vAIK$.
- (f) *Attestation response.* B delivers the signed message together with the SML and the $vAIK$ credential to T (2). Finally, B computes $K_{TB} = (g^t)^b \bmod p$.
- (g) T verifies whether the delivered credentials are authentic and verifies if all signatures are valid. Using the received $g^b \bmod p$, T computes $K_{TB} = (g^b)^a \bmod p$ and delivers an encrypted nonce in message (3) to B .
- (h) B decrypts the nonce with K_{TB} and delivers message (4) to T .

- (i) T decrypts message (4) and verifies freshness of the message. Finally, T verifies whether the platform configuration of B is trusted using the SML.
- (j) *Ticket creation.* T creates the attestation ticket shown in Equation 9.1 and delivers τ to B .

$$\tau = \{g^b \bmod p, g, p, \text{timestamp}\}_{K_T^{-1}} \quad (9.1)$$

- (k) B receives τ and creates a non-migratable TPM key K_{TPM} . This key is certified with the $vAIK$ and delivered to T . In addition, B chooses an arbitrary secret s and delivers it to T .
- (l) T encrypts s with K_{TPM} and a TTP specific symmetric key K_{TTP} . The encrypted secret is delivered to B in message (7).
- (m) B receives the encrypted secret $\{\{s\}_{K_{TPM}}\}_{K_{TTP}}$ and stores it.

After execution of the initialization phase, B is ready to execute the attestation phase. In the attestation phase, the local stored secret s is delivered to the TTP where it is decrypted and delivered back to B . In addition, T validates the platform configuration of B and delivers a ticket that vouches for the trust level of B . This ticket is later delivered to the server. The protocol run of the attestation phase is shown in Protocol 9.3.3.

Protocol 9.3.3: Attestation Phase

SUMMARY: B answers the attestation challenge of T

RESULT: Establishment of a secure attestation channel: unbinding of a secret and verification through the user

1. Protocol messages.

$$T \rightarrow B : Nt, g^t \bmod p, g, p \quad (1)$$

$$T \leftarrow B : \text{Cert}(vAIK, K_{vAIK}), \{Nt, PCR, g^b \bmod p\}_{K_{vAIK}^{-1}} \quad (2)$$

$$T \rightarrow B : \{Nb, g^t \bmod p\}_{K_{TB}} \quad (3)$$

$$T \leftarrow B : \{Nb, Nt, SML, g^b \bmod p\}_{K_{TB}} \quad (4)$$

$$T \rightarrow B : \{\{g^b \bmod p, g, p, \text{timestamp}\}_{K_T^{-1}}\}_{K_{AB}} \quad (5)$$

$$T \leftarrow B : \{\{\{s\}_{K_{TPM}}\}_{K_{TTP}}\}_{K_{TB}} \quad (6)$$

$$T \rightarrow B : \{\{s\}_{K_{TPM}}\}_{K_{TB}} \quad (7)$$

2. Protocol actions.

- (a) The protocol actions for delivering messages (1) to (5) are analogue to Protocol 9.3.2. See actions (a) to (j) of Protocol 9.3.2.

- (b) B loads the encrypted secret s from the background storage and delivers $\{\{\{s\}_{K_{TPM}}\}_{K_{TTP}}\}$ to T .
 - (c) T receives $\{\{\{s\}_{K_{TPM}}\}_{K_{TTP}}\}$ and if B is trusted, it encrypts this message and delivers $\{s\}_{K_{TPM}}$ back to B .
 - (d) B receives $\{s\}_{K_{TPM}}$ and decrypts the message using K_{TPM}^{-1} . The encrypted secret s is displayed to the user.
-

9.4 User Authentication and Ticket Presentation

Most current e-commerce applications use a password-based user authentication. This includes passwords, personal identification numbers (PINs), transaction authorization numbers (TANs) and more sophisticated one time passwords (OTP). These token-based user authentications are then integrated into an established TLS channel between client and server. However, one has to be careful about embedding these password-based authentication methods into an TLS-channel, since the confidentiality of the passwords for authenticating a user depends on the authenticity of the server. If the server authentication is done improperly by the user, e.g., by not clearly verifying the presented certificate and the URL (see [88], [4]), an adversary could masquerade as a server, collect usernames and passwords and later impersonate the user. This MitM attack is also possible if highly sophisticated one-time passwords or SecurID token-based authentication methods are used, since the adversary could directly replay the collected authentication data into an already established session to an authentic server. Since the client application does not prevent naive users from improperly verifying a presented server certificate (and transferring their password to a malicious server), additional security mechanisms must be in place.

9.4.1 Password-Based User Authentication and Ticket-Presentation

The solution we propose directly couples the user authentication to a second cryptographic channel inside the TLS session. This second channel fulfills two requirements:

1. proof-of-possession of a ticket bound to the client's platform issued by the TTP.
2. user-authentication is based on the already established secure channel and therefore only valid for this specific session.

Our proposed protocol is shown in Protocol 9.4.2. The protocol is executed between one e-commerce server, which we denote with A and a client B . In the first steps, an authentic TLS session is established and the ticket τ is verified. If the ticket τ is valid, the server computes a DH-key pair [46], which acts as challenge for the client.

Protocol 9.4.2: Ticket Presentation and User Authentication

RESULT: A authenticates the user

SUMMARY: Ticket presentation: ticket verification: key establishment with key-confirmation: user authentication

1. *Protocol messages.*

$A \leftrightarrow B : \text{Establish TLS channel} \quad (1)$

$A \leftarrow B : \{g^b \bmod p, g, p, \text{timestamp}\}_{K_T^{-1}} \quad (2)$

$A \rightarrow B : g^a \bmod p, \text{Valid} \quad (3)$

$A \leftarrow B : \{f(g^a \bmod p, g^b \bmod p, \text{password}), ID\}_{K_{AB}} \quad (4)$

2. *Protocol actions.*

- (a) A and B establish an TLS-channel. Alternatively, Protocol 3.7.4 can be used. In that case, protocol messages (2), (3) and (4) must be slightly adapted.
- (b) B transfers the Ticket $\tau = \{g^b \bmod p, g, p, \text{timestamp}\}_{K_T^{-1}}$ in message (2) to A .
- (c) A receives message (2), verifies freshness of the message and verifies all signatures. If the validation succeeds, A creates a random secret a , $2 \leq a \leq p-2$ and computes $g^a \bmod p$. A delivers message (3) to B .
- (d) B computes based on the ephemeral public DH-keys $v = f(g^a \bmod p, g^b \bmod p, \text{password})$. The *password* is a secret that, A and B have negotiated in advance, for example an arbitrary series of characters. The function f must be a collision resistant hash-function [54], so that an adversary is unable to compute the password based on the public keys and the output of f . Finally, B computes the shared key $K_{AB} = (g^a)^b \bmod p$ and transfers message (4) to A .
- (e) A computes $K_{AB} = (g^b)^a \bmod p$ and decrypts message (4). Based on the transferred public keys and the deposited password, A computes $v' = f(g^a \bmod p, g^b \bmod p, \text{password})$. If $v \stackrel{?}{=} v'$, the user of platform B with the identity ID has successfully authenticated himself. In addition, he provided a proof about the trust level of his platform and proven that he is in possession of the valid ticket τ .

Protocol 9.4.2 can be executed very efficiently, since the server only has to verify one signature and to compute the shared session key K_{AB} . From this point on, this shared key is used for the transaction. It should be noted that the generation of a new DH key-pair can be omitted if the server has already seen g and p and generated a key-pair based on these public parameters.

9.4.3 Authentication Based on Smart Cards

TLS can also be used to mutually authenticate user and server. For this purpose, all users are equipped with a certificate that is presented to the server. This certificate allows us to omit the additional authentication method accomplished inside an existing TLS session. This authentication mechanism prevents MitM attacks, since the adversary cannot access the private key, which is stored on the smart card and used for generating the TLS session key. If the user is in possession of a valid certificate and a smart card, a mutually-authenticated TLS channel is established into which the ticket T is transferred. In this case, steps 2-4 of our proposed Protocol 9.4.2 must be adapted to ensure that the platform is in possession of the ticket.

9.5 Security Analysis

This section provides an informal security analysis of our security mechanisms and the proposed protocols.

9.5.1 Forging a Trusted System Configuration

An attacker could try to forge a trusted system configuration by forging or replaying a valid ticket. Since a ticket is directly bound to the current attested platform configuration, an attacker cannot perform the challenge-response authentication process that confirms the key ownership, which is integrated into the ticket.

Another possible attack is to replace the existing TVM, including the STS, with a malicious TVM and simulate a successful attestation. For the success of this scheme, the adversary must let the user perform a new initialization phase. However, the users are able to detect this attack since they already completed the initialization phase and provided their STS with a secret.

9.5.2 Accessing Confidential User Data

An adversary who is trying to access the password of a certain user may attempt to simulate a successful attestation in order to force a user into entering his password into a malicious client configuration. To achieve this goal, an adversary needs access to s , which is only decryptable if both the TTP and the vTPM agree. A malicious host must, therefore, transfer s to an honest host under his control and decrypt the message by the TTP. Since the retrieved message is still encrypted with a key stored in the vTPM of the malicious host, the adversary needs to transfer the key back to the malicious host. However, the honest host runs a trusted system, which effectively prevents reading out the memory for accessing and retrieving the key.

An adversary could also try to access the symmetric encryption key K_{TTP} , which the TTP uses for encryption. Since the adversary can pick passwords at random, the adversary can perform an adaptive chosen-plaintext attack in attempt to crack the key K_{TTP} . We believe that this attack is very unlikely, since an adversary must pass the attestation phase and, therefore, cannot use additional tools to support his computation.

9.5.3 Security of the Password-Based User Authentication

Despite the existing shortcomings of password-based user authentication, e.g., their vulnerability to phishing attacks [4], this method is currently wide spread and most of the available e-commerce applications employ this method. Our proposed authentication scheme, which takes place in a second cryptographic channel, offers an end-to-end communication that ends in a trusted virtual machine. To authenticate as a valid user, one has to be able to answer the challenge, which is only possible if the platform has obtained attestation and uses the corresponding ticket. The current one-time-password (OTP) for authenticating the user is then computed based on the public key from the server, the public key of the client and the shared password. A MitM may relay the challenge ($g^a \bmod p$) to the authentic user, but he cannot compute the shared key and, thus, is not able to manipulate the upcoming transactions.

To clarify this attack, we look at the attack vector in detail. We assume that an adversary M tries to authenticate himself against the server A using the obtained OTP from B . The transmitted protocol messages are shown in Figure 9.3. The assumption for this attack is that the adversary M is in possession of a ticket ($g^i \bmod p$, g , p , *timestamp*) that vouches for the trust level of his platform. In addition, we assume that the user of platform B improperly verifies only the TLS-certificate presented to him.

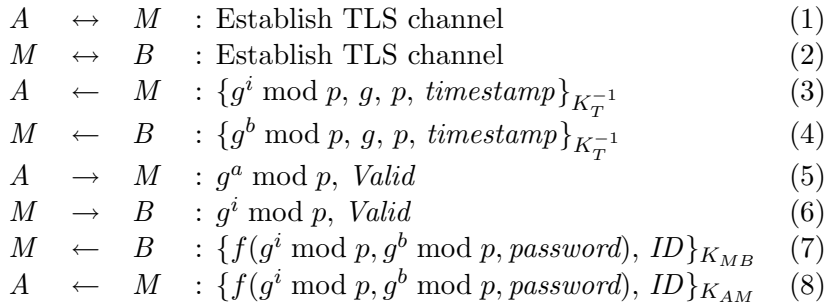


Figure 9.3: Man-in-the-Middle Attack on Protocol 9.4.2

For a successful attack, it is necessary that the attacker modifies the challenge sent to him by the server in step (5). Otherwise, it would not be possible for him to decrypt the following communication, since A and B would have negotiated a cryptographic key K_{AB} . Due to the modified challenge in message (6), B computes his OTP based on this message and transfers it to M . The adversary is able to encrypt this message since he is in possession of the key K_{MB} , but he cannot use this in another challenge established between the adversary and A .

To prevent an adversary from launching dictionary attacks or brute-force attacks, the public key of the client is additionally included in the computed OTP. Without injecting the public key, a malicious server, trying to crack many passwords simultaneously, only needs to guess a password once, compute the OTP, and compare it to all the other OTPs. In that case, the malicious server uses the same public key as a challenge for multiple clients. However, with the public key of the client, all OTPs are

based on different public keys of the client; so each guess must be computed separately for each public key. We suggest that the password be a long secret in order to increase the computational overhead for a malicious server to perform attacks of this type.

The robustness of this authentication scheme relies on the security of the one-way hash-function used to compute f , more specifically, on the pre-image resistance of the hash function [131]. f must be appropriately chosen so that it is a cryptographic hash function which is collision and pre-image resistant. If these conditions are satisfied, the probability of finding the password is comparable as hard as finding the pre-image of an input without additionally adding public Diffie-Helman parameters. This is, for example, easily verifiable if f is a random oracle [18].

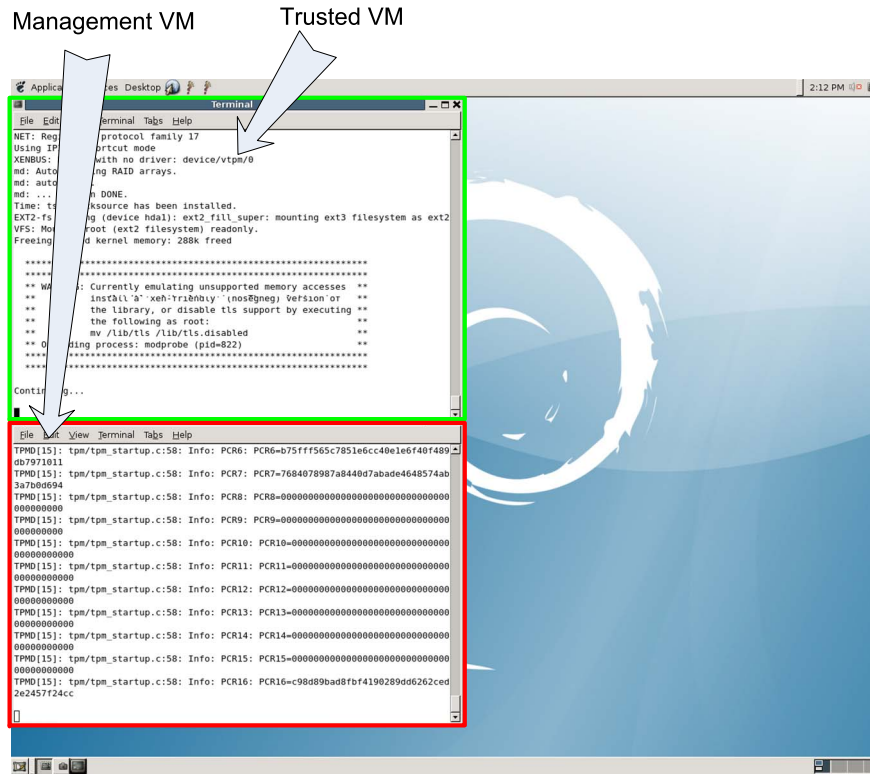


Figure 9.4: The spawning of a Trusted VM. The upper shell shows the bootstrap procedure of the TVM. The lower shell shows the contents of the vTPM. In this realization, the TVM uses the x-server of the Management VM.

9.6 Implementation

We have implemented our proposed e-commerce architecture, including a trusted third party, secure transaction software and a trading entity. The trading entity is a conventional shop system that has been modified to support the creation and verification of

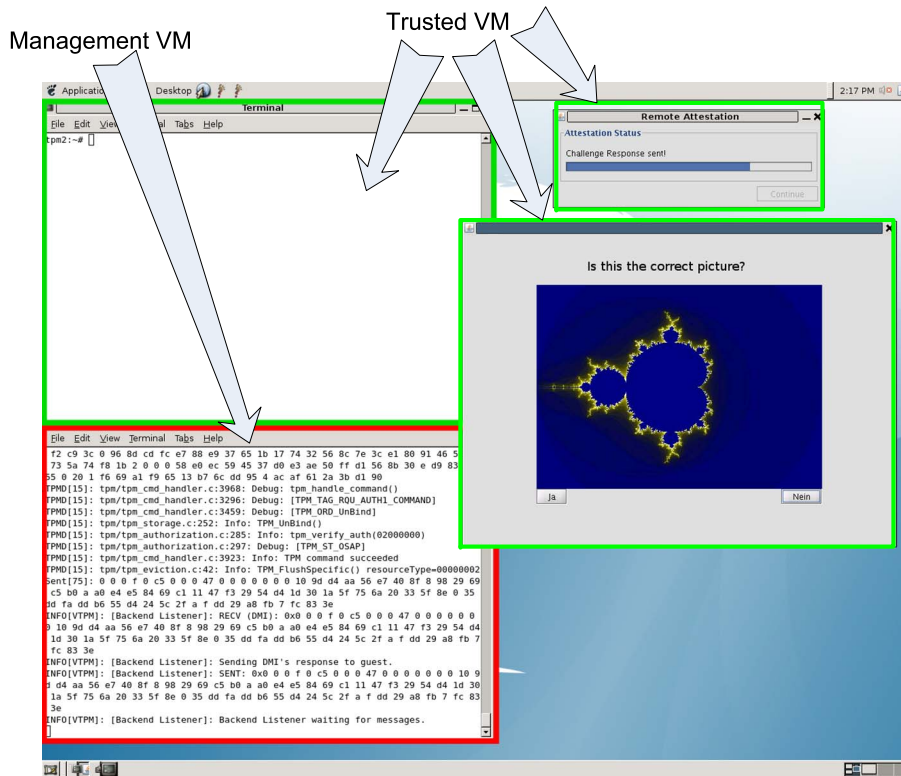


Figure 9.5: Bootstrap procedure of the STS. After starting the TVM, the STS is directly executed. The picture shows the user’s stored secret as deposited inside the TVM. The shell also shows the operations of the vTPM-Manager running in the Management VM.

digital signatures and to verify the authenticity and validity of the ticket that vouches for a trusted client configuration. Since the ticket issuing and configuration verification is mainly done by the TTP, the extensions to the conventional shop system are very minimal.

The STS is implemented in Java and is able to interpret any website written in HTML and transferred using HTTP; however, it only provides the full security mechanisms if it accesses a web-server that has been enhanced with our security mechanisms. It is able to carry out secure transactions between a trading entity and implements our proposed protocols. The STS runs on a Linux kernel that has been modified to use the security mechanisms of the underlying security architecture. We removed unnecessary device drivers and configured the STS and the Linux kernel as a virtual appliance that is compatible to our security architecture introduced in Chapter 7. Figure 9.4 shows the bootstrap procedure of a TVM.

The secure transaction software implements those protocols proposed in this chapter that establish a secure attestation channel and is able to make trust visible to the user. After starting the secure transaction software, the stored secret is displayed to the user as specified in Protocol 9.3.3. This process is shown in Figure 9.5. After validating the

secret by the user, a secure connection to the trading entity can be established and the secure transaction software displays the entry page of the e-commerce server (Figure 9.6).

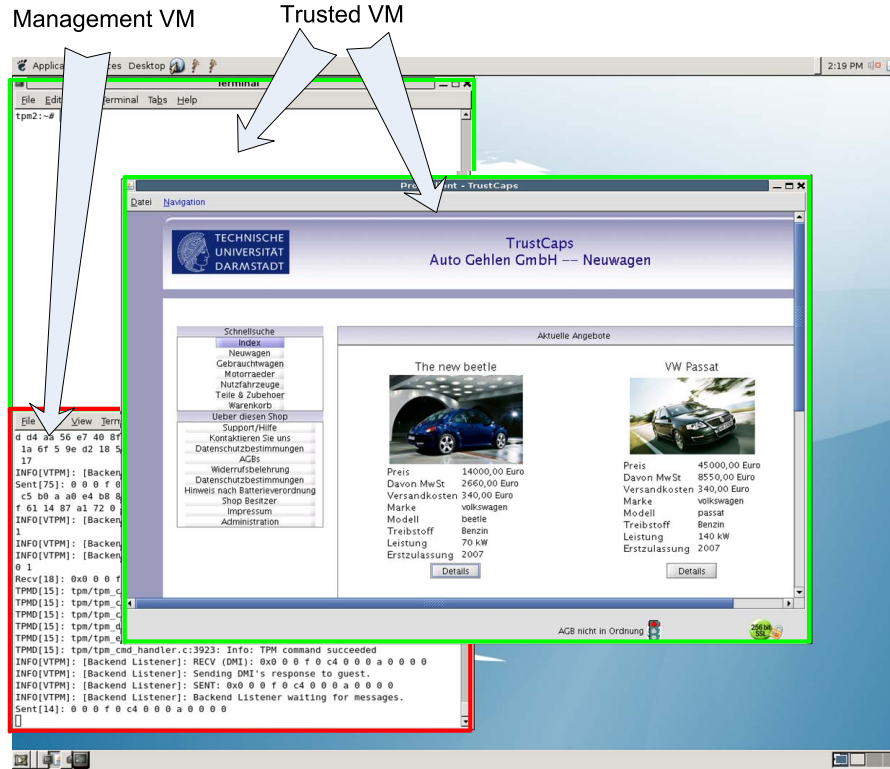


Figure 9.6: Started STS. After successful validation of the platform configuration, the TVM starts the STS.

Due to the existing shortcomings of password-based authentication methods, we only implemented smart card-based authentication mechanisms that take place inside the second cryptographic channel. To ensure that the STS is in possession of a valid ticket, we adapted steps 2-4 of the protocol shown in Protocol 9.4.2 and performed a slightly different challenge-response authentication process, namely a process that does not transfer the password and the user-ID. The user is identified and authentication is based on the information given on the smart card. For this purpose, we integrated a smart-card device driver inside the virtual appliance of the Trusted VM and directly accessed the USB-port of the relevant smart-card reader using *pciback* scripts [189].

9.7 Related Work

Oppliger et al. [112] identified the problem of man-in-the-middle attacks in TLS sessions and proposed a solution to this problem. The main idea of their work is to base the user authentication on the user's secret and on the state information of the TLS channel. The

disadvantage of this solution is that it depends on the trust level of the client application and also requires that the user possess a secure ID-token with a small display.

Another interesting solution to secure e-commerce is proposed in [154]. Like us, Singarevelu et al. propose reducing the code size and establishing isolated environments using a microkernel. However, in contrast to our approach, Singarevelu et al. do not show how attestation techniques could be used to provide assurances that a user is still acting with a trusted environment. In addition, the approach presented by Singarevelu et al. does not provide any means of preventing man-in-the-middle attacks on TLS.

There has also been much work done to prevent phishing attacks in electronic commerce. The objectives, motivations and attack patterns of phishers have been extensively studied in the literature (e.g., [72, 44, 111]) and many proposals have been made to prevent these attacks (e.g., [111, 188]).

In this context, one very interesting proposal has been made by Gajek et al. [65]. The authors proposed to augment a web-browser with an additional wallet that is responsible for performing user authentication and for authenticating websites. The wallet is strictly isolated from the web-browser and runs in a compartment established by the PERSEUS security kernel [121]; it is, thus, robust against infection from malware. However, since the proposal does not support attestation techniques, it does not provide any means of providing a user with assurances that their used web-browser or the wallet is still trusted.

Cox et al. [40] propose the Tahoma Web browsing system. This architecture utilizes virtualization techniques to execute multiple isolated compartments on the Xen-hypervisor. In each compartment, an instance of a full-fledged web-browser runs and a network proxy running inside the management domain of Xen [14], controls the network connection. While the approach is able to restrict the access rights of one browser instance based on a specified policy, it is not able to handle and prevent man-in-the-middle attacks on TLS. In addition, there are no mechanisms present to enable a user to place trust into a web-browser running in an isolated compartment.

Other work that looks at trusted computing and e-commerce includes [12, 13, 4]. However, these approaches do not show how a secure e-commerce architecture that supports attestation could be realized. Thus, these approaches are not able to generate assurances of the trust level of a user's platform or to transfer these assurances to the user.

9.8 Summary

Server platforms are relatively robust and, whilst capable of being compromised, are typically not the easiest target for cyber-criminals. Instead, client platforms are increasingly being subverted since users typically do not have the same financial motivation (as merchants) to harden their systems. Indeed, many users neither know how to harden their systems nor how to discover if their system has been subverted. The solution we have proposed is based on virtualization in combination with attestation techniques and allows a user to ensure that a particular client configuration has not been tampered

with and is trusted for the duration of the transaction. In addition, our architecture ensures that confidential data, such as authentication passwords, are not accidentally transferred to malicious servers that masquerade as authentic servers.

Chapter 10

Signature Creation with TPMs

In this chapter, we discuss the potential of the TPM and give considerations as to whether a TPM could be used as a secure signature creation device. We examine whether the TPM can be used as a secure signature creation device that conforms to the EU Electronic Signature Directive as well as to the German Electronic Signature Law. In addition, we argue that if the TPM can be used as secure signature creation device, a trusted signing software also becomes necessary. This chapter shares some material with *The Creation of Qualified Signatures with Trusted Platform Modules* [162] and *Erzeugung elektronischer Signaturen mittels Trusted Platform Modules* [163].

10.1 Introduction

Despite the existing shortcomings of password-based user authentication, i.e., their vulnerability to phishing attacks [4], these techniques are currently wide spread and most of the available e-commerce applications are based on these techniques. To overcome these shortcomings qualified electronic signatures might be used. These signatures are considered to be a vital component for e-commerce transactions in the future, since they are capable of linking a particular user to a particular transaction effected over the internet. In contrast to the password-based user authentication, they can act as a *prima facie* evidence in trial since they offer non-repudiation, which is especially relevant in the realm of e-commerce. Thus, qualified signatures are a very important component for trust-establishment.

A qualified electronic signature that confirms to article 5(1) of the EU Electronic Signature Directive (EU Directive) and § 2(3) of the Signature Act 2001 (SigG) is an advanced signature as specified by article 2(2) of the EU Directive and § 2(2) of SigG, which is based on a qualified certificate and which is created by a secure signature creation device.

However, qualified signatures are not widely used, and it is asserted that the failure to use qualified signatures thereby deprives the market for electronic commerce of an important source of potential growth [126]. A promising approach to overcome this shortcoming is to use the Trusted Platform Module (TPM) to create a qualified signa-

tures. The TPM is already available in approximately 200 million personal computers [178] and this technology is supported by many hardware vendors and is therefore widespread. One of the properties required of this chip is to perform the necessary cryptographic functions to create an advanced signature. However, the ability to perform the required mathematical operations is not enough to use this device to create an advanced signature. This is due to the requirements for a compliant device for the creation of an advanced signature, as set out in article 5(1) and Annex I - III of the EU Directive and the Signature Act 2001.

Unfortunately, even if we were to assume that the TPM is able to create a qualified signature, the problem arises that the software that is responsible for communicating with the TPM might tamper on data delivered to the TPM. As a result, the owner of the TPM might sign false or malicious transaction data without noticing. This may result in the fact that the signatory made an unintentional contract which he cannot deny. As a result, attestation techniques become necessary.

In this chapter, we will firstly validate whether the TPM can be used to create qualified electronic signatures, and if so, whether such signatures could be used in e-commerce. Thereafter, this chapter considers whether the TPM fulfills the technological requirements of article 5(1) of the EU Directive and § 2(3) SigG, together with the criteria provided for in Annex III of the EU Directive and § 17 Abs. 1 SigG, as well as § 15 Signature Decree 2001 (SigVO).

10.2 Testing SigG Conformity of the TPM

In this section, we evaluate whether the TPM can be used as a secure signature creation device that conforms to the EU Electronic Signature Directive as well as to the German Electronic Signature Law. For this purpose, we examine whether the mechanisms provided by the TPM meet the basic requirements of the signature acts.

10.2.1 SigG Conformity of the TPM

In order to create a qualified electronic signature, the signature must pass the requirements specified in article 5(1) of the EU Directive and § 2 No. 3 SigG. According to these provisions, the signature must be created with a secure signature creation device. The detailed requirements are defined in annex III of the EU Directive and for Germany in § 17(1) SigG and § 15(1) and (5) SigVO. According to the provisions of annex 1(I)(1) of SigVO, the secure signature creation device must fulfill the assurance level EAL 4 and be tested against an attacker with a high attack potential (strength of function high). As already shown, this requirement is not covered by the TPM specification, but can be fulfilled by the TPM products. According to the provisions of annex III(1)(b) of the EU Directive and to § 17 (1) SigG, the signature creation device must reliably detect a potential forgery or a modification of the signature. Annex III of the EU Directive provides as follows:

Requirements for secure signature-creation devices Secure signature-creation devices must, by appropriate technical and procedural means, ensure at the least that:

1. the signature-creation-data used for signature generation can practically occur only once, and that their secrecy is reasonably assured
2. the signature-creation-data used for signature generation cannot, with reasonable assurance, be derived and the signature is protected against forgery using currently available technology
3. the signature-creation-data used for signature generation can be reliably protected by the legitimate signatory against the use of others

A forgery or a modification of a signature is recognizable if the verification mechanism of a digital signature cannot be bypassed or deactivated. Forging a signature is not detectable if an unauthorized person can obtain access to the signature key. This situation could occur in the following cases:

- The same signature pair is generated multiple times
- The private key used for signature creation can be obtained through the public key
- The private key can be guessed
- The private key is copied during generation and transferred to another unauthorized person
- The private key is accessible or useable in a stolen or found signature creation device

The TPM uses a physical random number generator for key generation, which causes the distribution of all keys to have an equal probability. In combination with keys consisting of 2048 bit, every key is unique with sufficient probability. The key length used by the TPM is appropriate for signature keys until 2011 [25] with respect to the implemented hash function, which is, despite the SHA-1 weakness, applicable until 2009. The key length means it is almost impossible to guess the private key, and the RSA algorithm provides assurance that the private key cannot be computed based on the public key. The requirement that the keys must not be revealed according to § 15(1)(2) SigVO (tamper-resistance) is not fulfilled by the TPM specification. However, it is suggested that the TPM products meet that requirement. The TPM also protects the signature key against the use by others through the owner-password, therefore fulfilling requirement (c) of the EU Directive. The method of generating a secure owner-password is discussed in the following section.

10.2.2 Results

The TPM specification meets the basic requirements of annex III of the EU Directive, but does not meet all formal requirements of the Signature Act 2001. The available products fulfill all requirements of the German signature law from the technical perspective. Therefore, these products can be validated according to annex 1 of the SigVO24 and approved as secure signature creation devices.

10.3 Qualified Electronic Signatures

As already described in the preceding section, the TPM provides the functions to create an advanced signature. Since the TPM was originally designed to carry platform-specific data and information about the user, it must be determined whether it can also be used for this purpose.

10.3.1 Identification

The basic difference between the smart card with the ability to create an advanced signature and the TPM, is that a smart card is directly bound to a certain user, while the TPM is shipped without personal certificates. In order to use the TPM to carry information relating to the user, it is therefore necessary that a certification service provider (CSP) identifies the owner of a TPM and certifies the corresponding public signature key. The applicant for a qualified certificate must therefore, according to the provisions of annex II(d) of the EU Directive and § 5 (1)(1) SigG, be clearly identified by the CSP, for example, by validating their identification card, if such a form of identification is acceptable. The user can physically visit the CSP for this purpose, or a third party may, in accordance with the provisions of § 4(5) SigG, validate the identity of the user. The German PostIdent method run by the German Post AG may be a useful method to use for this purpose. This method is still used by many on-line credit institutes to identify their customers according to the provisions of § 154 Tax Law. The identification can therefore be accomplished in the post office or at the applicant's home. After the identification process is complete, the applicant receives a unique secret, which is used to assign the signature key to a person. This secret must be transferred over secure channels to the applicant. This can be done by certified mail or physically handed over to the applicant in person.

10.3.2 Issuing Qualified Certificates

In Germany, there are additional requirements to be fulfilled when issuing a qualified signature, although similar problems exist under the terms of the EU Directive which are described in article 2(10) of the EU Directive. According to § 5(6) and § 15(7)(2) SigG, the certification service provider must ensure that the applicant that wishes to obtain a signature is in possession of a secure signature creation device - in this case, the TPM. Qualitatively, both requirements of § 5(6) and § 15(7)(2) SigG are identical.

For this purpose, a protocol is introduced, illustrated in Figure 10.1, that fulfills the requirement of proving the possession of a secure signature creation device, and also ensures that the keys are placed into this specific device. One requirement for the success of this scheme is that the CSP offers a platform that enables secure communication between applicant and the CSP, by using the Transport Layer Security. In this context, it is necessary for the CSP to authenticate itself against the applicant. It should be noted that the applicant must use special software that implements the protocol described in Figure 10.1. This software should validate the certificate and cancel the connection if the certificate is not valid. This is essential, because it is possible that a false person masquerading as the legitimate CSP may present the applicant with what seems like a legitimate certificate to authenticate themselves. Figure 10.1 shows the CSP and the platform of A (the applicant). The applicant's platform A is split into two components: the TPM, which carries out the cryptographic functions, and a certificate issuing software, which is provided by the CSP.

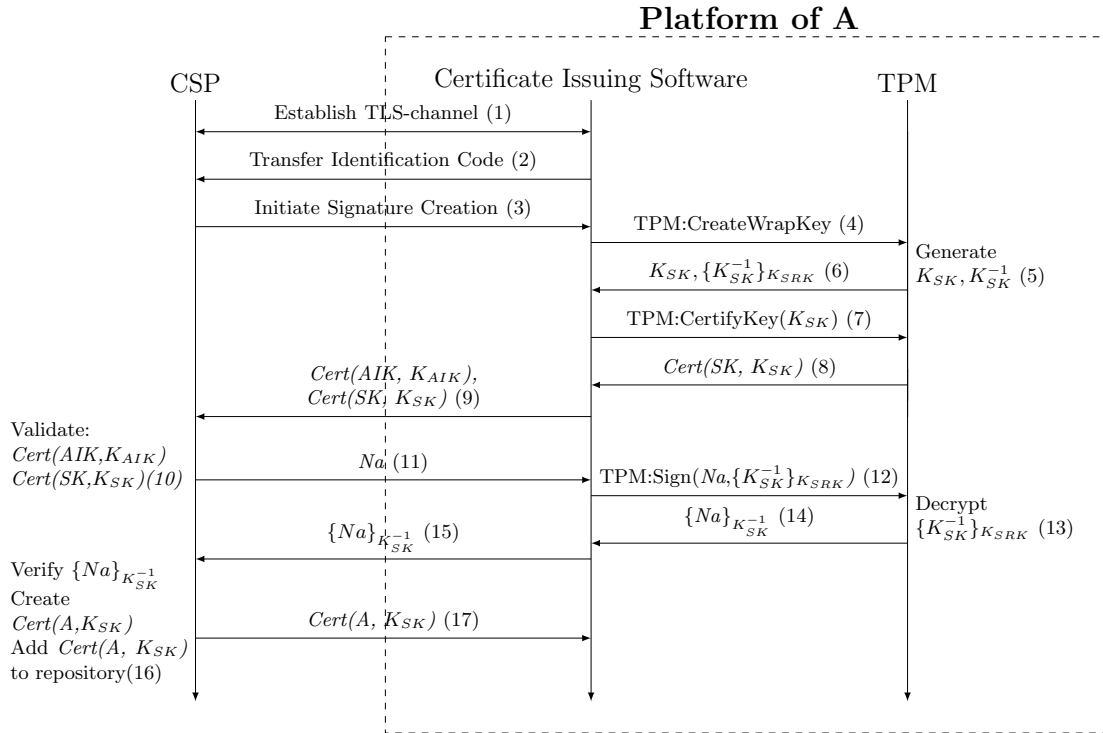


Figure 10.1: Protocol for certificate issuance

After the applicant (A) has entered their personal data, they transfer their identification code to the CSP using the certificate issuing software (steps 1 - 3). In the next step, the TPM of A generates a signature key pair K_{SK}^{-1} and K_{SK} using the physical random number generator of the TPM. In accordance with the provisions of § 17(1)(3), this key can be generated on a secure signature creation device - namely the TPM,

and does not need to be added externally. This also acts as a form of self-protection, since this guarantees that the key is protected over its lifetime and is not exposed to third parties. After the private key pair is generated, it is then encrypted with the *SRK* (step 6 in Figure 1) of the TPM and returned to the user. The public part of the signature-key is signed with an Attestation Identity Key (AIK) of the TPM (7). This certification proves that the corresponding key-pair is held in the protected storage of a valid TPM and that the signed key is a key that cannot be moved. Identity keys can only sign data that originates from the TPM. Thus, a valid signature of data proves that the data was generated on a TPM and is protected by the TPM's secure storage, in that it is part of the key-hierarchy of the *SRK*. The identity key is only valid if the TPM has not been tampered with and the TPM is authentic. In the next step (8 and 9), the TPM transfers the certificate of the signature key ($Cert(SK, K_{SK})$) and the certificate of the identity key $Cert(AIK, K_{AIK})$ to the CSP. The CSP then verifies the authenticity of the signatures and verifies whether the *AIK* is a valid identity key (10). If the verification succeeds, the CSP has confirmation that K_{SK}^{-1} is held in the protected storage of a genuine TPM. The CSP then verifies in steps 11-16 the freshness of K_{SK}^{-1} and whether the applicant has access to the signature key by requesting a test signature (proof of possession). To create a successful signature, the TPM must decrypt the encrypted K_{SK}^{-1} . This is only possible if the correct *SRK* is stored inside the protected storage of the TPM, and the owner has delivered the correct password for decrypting the *SRK*. This kind of sample signature also fulfills the control duties of the CSP. Since the applicant must perform a sample signature, it is guaranteed that the applicant possesses the required knowledge (password) for a signature creation and that this unit is under the control of the person using the password. Finally, the CSP signs the public part of the signature key and issues the corresponding certificate, which it adds to his own directory service. This certificate is then transferred to the applicant (17).

10.4 Trusted Software

In order to use the TPM as qualified signature creation device, the software that executes the initialization protocol and the software that is responsible for communicating with the TPM need to be trusted. We will discuss this issue in this section.

10.4.1 Trusted Initializing Software

It is necessary to use an initializing software that reliably enforces the protocol set out above. This software is also responsible for supporting the user during the creation of a secure password. In contrast to smart cards, which often include the use of a counter to prevent more than a set number of attempts to correctly guess the password, the TPM does not provide such a function. The TPM specification requires a mechanism that prevents dictionary attacks, but the specification is not specific on this point, and leaves this to the TPM vendor. As a result, different products exist which differ in their implementation. For example, the STM TPM chip provides a counter to prevent more than

a set number of attempts to correctly guess the password. In contrast to smart cards, which prevent further use of the card after several incorrectly entered passwords, the STM TPM chip only increases the reaction time after 15 false attempts. The generation of a secure password should take place before a signature-key is generated, to ensure that the signature-key is protected by a secure password and to prevent the use of a chip containing an insecure password. The initialization process is important, which means that the software must be trusted. This could, for example, be performed by a boot CD that has been extended by the functions set out above. This software is configured so that it only supports connections to a specific CSP and prevents remote access to the underlying hardware TPM. The CSP must also validate whether the software used for the acceptance of the initial signature reliably enforces this requirement, because it is possible for the applicant to place their signature key into a TPM, which they can only obtain access to remotely. To ensure that a human being has physical access to the TPM, the CSP uses the integrity measurement and reporting functionality provided by the TPM. The initialization software is pre-configured in such a way that the TPM measures all running software components and attests this system state to the CSP. The CSP can then decide via a secure attestation channel, for example Protocol 3.7.2, whether a trusted initialization software is used. If this verification is successful and the protocol shown in Figure 10.1 has been executed, the CSP issues the corresponding certificate and signs the applicant's public key.

10.4.2 Trusted Signing Software

Besides the trusted initialization software, a trusted signing software is necessary that prevents that the transaction data that is to be signed is maliciously modified before the data is passed to the TPM. A trusted signing software displays the data that is to be signed to the user and passes the data to the TPM.

Both trusted initialization software and trusted signing software should be realized as a virtual appliance which is executed in a Trusted VM. These software components then run upon the security architecture proposed by us in Chapter 7 where the integrity and the trust level of this initialization software is assured. The trust level of these software components can be ensured by using the protocols introduced in Section 9.3.1. However, it should be noted that in that case the TPM architecture of Chapter 8 must be used in order to satisfy the requirement of providing lifetime protection of cryptographic secrets. This requirement is for example not satisfied by software TPMs which cannot act as a secure signature creation device.

10.4.3 Knowledge and possession

As described in the preceding sections, the TPM offers the possibility to store and use personal certificates. But it must still be determined whether the signatory is also in direct possession of a secure signature creation device, and how this fact can be

proven to the CSP¹. The EU Directive requires, pursuant to annex III (1)(c), that the signature key can be reliably protected by the legitimate signatory against the use of others. The provisions of § 15(1) SigVO state, in more detail, that it should only be possible to obtain access to signature keys after the identification of the applicant on the basis of knowledge and possession, or by means of a measurement of a biometric attribute. Since biometric attributes do not provide the same level regarding security, knowledge and possession must be used as the means of identification. The requirement that it should only be possible to obtain access to the signature key if the applicant is successfully identified on the basis of knowledge and possession, provides a reasonable level of assurance that the signature was created by a specific person. Since neither knowledge nor possession on its own are reliable attributes to identify somebody fully, both attributes must exist simultaneously. Only when both attributes are used together, it is argued, can an electronic signature replace a manuscript signature and therefore act as *prima facie* evidence in trial.

One issue is the meaning of possession in the context of a TPM. The legislation covers smart cards as secure signature creation devices [134]. However, the official statement also mentions special components as secure signature creation devices to be used in mainframe architectures [26]. Since the definition of a secure signature creation device is not very specific, it is not necessary for the secure signature creation device to be portable, as provided for by the provisions of § 15(1) SigVO. The aim of the regulation is to ensure the signatory has the secure signature creation device in their custody, and is capable of preventing unauthorized access to the device. In the case of mobile devices, this custody can be adduced by physically inspecting the device. Unfortunately, it is more difficult to prove that the signatory has sole access to the device and can prevent unauthorized access. In this context, the TPM could be situated in an external environment, and the signatory could obtain access to this device remotely, and protect it with the password. This fact would neither fulfill the requirement of §§ 5(6) and 15(7) SigG nor the protection purpose of § 15(1) SigVO. Therefore, the CSP must verify whether the applicant of an electronic signature has the secure signature creation device in their custody and is capable of preventing unauthorized access. To achieve these properties, it might be necessary for an employee of the CSP to physically attend the applicant's premises to confirm possession. The use of a password implies that the signatory has control of the secure signature creation device. This does not exclude the creation of signatures on remote devices, if the signatory can prevent unauthorized access to the signature creation device by protecting it physically.

10.5 Summary

The Trusted Platform Module offers, from the technological point of view, the possibility to use and store personal certificates and their corresponding keys. In the realm of the

¹It should be noted that the protocol described in this article only proves that the generated keys are generated on a secure signature creation device, and not whether the applicant is really the one who initiates the protocol.

EU Directive, it can serve as a secure signature creation device. In Germany, to create qualified signatures with the TPM, the TPM must be approved as a secure signature creation device according to § 17(4) SigG. Furthermore, the TPM must be protected against unauthorized access and be in the signatory's direct possession and secured with an additional method of proving possession. If the TPM is to be used as a signature creation device, it is necessary that the TPM implements the protocol set out in this chapter, which enables a qualified certificate to be issued to a specific TPM signing key. In addition, it is required that the initialization and signature software is protected from tampering which can be ensured using attestation techniques. Based on the high availability and low cost of a TPM, it can reduce the costs involved in creating signatures and possibly act to increase the use of secure signature creation devices.

Part V

Summary

Chapter 11

Conclusions

In this thesis, we developed concepts and methods to leverage attestation techniques for trust establishment in distributed systems. To achieve this goal, we first analyzed the challenges that need to be solved in order to use attestation techniques for trust-establishment. We identified building-blocks which need to be used to overcome the existing challenges. Our identified building-blocks can be used to establish trust between distributed systems that communicate over an insecure communication channel. The identified three building blocks are as follows:

Attestation Protocols Attestation protocols enable a remote entity to validate the trust level of another entity by verifying that the software loaded on a system corresponds to the expectations. Since attestation protocols are a key-mechanism for securing a system, they have to be secure. We have extracted three requirements which must be satisfied in order to design secure attestation protocols:

1. Attestation techniques must always ensure freshness and authenticity of integrity information.
2. The explicit attestation must be performed over secure channels.
3. Attestation protocols must integrate a key-establishment component that ensures that the established attestation channel is authentic.

We have shown that the TCG-defined attestation protocol does not satisfy all our extracted requirements and is, thus, vulnerable to a masquerading attack. We also proposed a secure attestation protocol that satisfies all our extracted requirements and can, therefore, be used to securely validate the trust level of a remote entity. The resulting channel that is established by a secure attestation protocol is referred to as secure attestation channel. We formally verified the secure attestation protocol to validate whether our pre-defined security goals are satisfied. We also implemented our attestation protocol and analyzed its performance to show that our proposed secure attestation protocol is efficient.

In scenarios where an entity is frequently requested to deliver integrity information, existing attestation protocols scale badly. Such a scenario is, for example, a classical client-server architecture, where a large number of clients frequently request integrity information of a particular server. This bad scalability is caused by the fact that a Trusted Platform Module (TPM) possesses very limited computation power and is highly involved in the process of platform attestation. To solve this problem, we presented solutions to overcome the bottleneck of a TPM and therefore improve the scalability of attestation protocols. All our proposed protocols satisfy our identified requirements of a secure attestation channel and, thus, allow a remote entity to securely validate the trust level of another entity. We also presented a performance evaluation as well as a security analysis of our proposed protocol; the results clearly indicate that the protocols can considerably reduce the performance overhead of the attestation process.

One critical component to verify the integrity of a remote platform configuration is the Stored Measurement Log (SML). The SML is a data structure in which all events that were measured by the TPM are reported. The SML is needed for the verifying entity to make a trust decision about the requesting system's platform configuration. However, since this approach reveals the platform configuration of the requesting system, it is not always a desirable feature. To overcome this limitation, we proposed a distributed integrity validation architecture that allows outsourcing the attestation process. The architecture enables dividing a whole platform configuration into self-contained, independent parts. This approach allows to make a statement of the trust level of a platform even though the real platform configuration system is camouflaged. We also presented a security protocol which realizes this concept and satisfies our identified requirements of a secure attestation channel.

Attestation protocols typically require asymmetric cryptography and a relative high computation power to compute and establish cryptographic keys that subsequently realize a secure attestation channel. These protocols are of intrinsic complexity as a validation entity has to determine the trust level of a particular communication partner based on the complex SML and the received PCR values. However, in resource-constrained systems, such as wireless sensor networks or embedded systems, a complex attestation process which is based on asymmetric cryptography is impractical. To solve this problem, we proposed two lightweight TPM-based attestation protocols for resource constraint systems. These protocols allow performing an *implicit attestation* by utilizing the fact that the software configuration of resource constraint systems often do not change during their whole lifetime. Our proposed protocols enable a low-cost node to verify the trust level of another node that possesses more computation power and is equipped with a Trusted Platform Module. Our proposed protocols do not require expensive public key cryptography and the exchanged messages are very short. We evaluated the performance of our proposed protocols showing that both the overhead for storage and the energy consumption are negligible.

Security Architecture We analyzed the requirements and challenges that need to be solved to integrate attestation techniques into current available operating systems.

In this context, we identified three main problems that prevent a successful usage of this technology (*time discrepancies*, *incomplete measurements*, and *inefficiency*). The time discrepancy problem indicates that the TPM-based load-time measurement does not correctly reflect the runtime-behavior. The problem of incomplete measurements indicates that not all executed system components that affect the platform integrity are measured. The third problem occurs because the approach is basically inefficient, since it requires all measurements to be known and fully trusted, even if they do not have a direct impact on a specific target application. To overcome these problems, we presented the design and implementation of a security architecture that is based on virtualization. Using this approach, we can establish isolated trusted environments which allow overcoming the inefficiency and time discrepancy problem. We also presented a new approach for measuring execution environments which solves the problem of incomplete measurements. The security architecture enables (1) leveraging attestation techniques for trust-establishment by achieving a meaningful attestation process and (2) containing security breaches to an isolated compartment. This security architecture can be used as a foundation to build more secure and reliable operating systems.

In our proposed security architecture, we used virtual TPMs to make the functionality of a hardware TPM available inside an isolated trusted execution environment. Virtual TPMs possess the full functionalities of a hardware TPM, however, they do not possess the same security mechanism. As a matter of these facts, we presented the concept and design of a virtualization-enhanced hardware TPM that is able to be used inside an isolated trusted execution environment and possesses the same security mechanisms as a hardware TPM. Using our proposed hardware TPM, it is possible to relinquish our security architecture from the need of using virtual TPMs. The virtualization-enhanced TPM utilizes recent developments in the virtualization technology of processor architectures and can, thus, easily be adapted to integrate trusted computing technology in next-generation processor architectures. In that case, highly-efficient and secure trusted computing technology would be available to next-generation operating systems that are based on virtualization.

Secure Transaction Software A secure transaction software is our third building-block to establish trust into a remote or a local system. This software typically implements the attestation protocols and runs on our attestation-supporting security architecture. In combination with the first two building-blocks, the secure transaction software is robust against infections from malware and provides basic primitives to establish trust into a remote or a local system. The purpose of a secure transaction software is to transfer trust to the user by making trust visible. To this end, we have adapted and extended one of our proposed attestation protocol to provide the user of a specific platform and a remote platform with assurances that a specific platform has not been tampered with and is trusted for the duration of a sensitive transaction.

Qualified signatures are a vital component for trust establishment since they offer non-repudiation, which is especially relevant in the realm of e-commerce. However, qualified signatures are not widely used, and it is asserted that the failure to use qual-

ified signatures deprives the market for electronic commerce of an important source of potential growth. To resolve this problem, the Trusted Platform Module which is already integrated in many systems could be used. We discussed the potential of the TPM and gave considerations as to whether a TPM could be used as a secure signature creation device. We have shown that, from the technological point of view, the TPM is able to use and protect personal certificates and their corresponding keys. In the realm of the EU, it can, thus, serve as a secure signature creation device. Based on the high availability and low cost of a TPM, it can reduce the costs involved in creating signatures and possibly act to increase the use of secure signature creation devices.

Drawn conclusions In short, based on our building-blocks, the following conclusions can be drawn:

- Using attestation techniques for trust establishment requires that some sort of secure attestation channel is established.
- In order to use the TCG-defined platform attestation in highly-frequented scenarios, the protocols to perform platform attestation must be adapted.
- To satisfy privacy issues of the TCG-defined platform attestation, a privacy-protecting mechanism must be in place that camouflages the real platform configuration.
- A secure attestation channel can also be established in systems with very low computational power. This enables providing trust assurances in these systems.
- The TCG-defined platform attestation possesses intrinsic problems causing the attestation not to be usable in system with high computation power. Virtualization techniques are a sound way for overcoming these intrinsic challenges.
- To securely use platform attestation in virtualization-enhanced operating system environments, a virtualization-enhanced hardware TPM is required.
- If attestation techniques are used to provide the user of a platform with trust assurances, a secure transaction software is necessary which is able to make trust visible.
- The TPM can be used to create qualified signatures, making the creation of qualified signatures highly available.

Part VI

Appendix

Appendix A

AVISPA Sources

In this chapter, we present the AVISPA source code for a number of selected protocols which we modelled.

A.1 Protocol 3.6.3

PROTOCOL: Enhanced Robust Integrity Reporting Protocol

PURPOSE: Enhanced-IRP is intended to provide secure integrity reporting.

The protocol proceeds between a verifier (role Alice) and a TPM-enhanced client (role Bob). The verifier wants to securely validate the platform configuration of the client. We assume the existence of one certification authority with public key K_s . This certification authority also created the AIK-Credential. In essence, K_s also acts as privacy CA. The agents possess certificates of the form $\{X, K_x\}_{\text{inv}(K_s)}$.

```
role alice (A, B: agent,
            Ka, Ks: public_key,
            H: function,
            G: text,
            SND, RCV: channel (dy))
  played_by A def=
    local State : nat,
      Na, Nb: text,
      X:text,
      Kb: public_key,
      SML: text,
      PCR:text,
      SK:message,
      KEr:message

    init State := 0
```

```

transition

0. State = 0 /\ RCV(start) =|>
   State' := 2 /\ Na' := new()
               /\ X' := new()
               /\ SND(Na'.exp(G,X'))

2. State = 2 /\ RCV({B.AIK}_inv(Ks).{Na.KEr'.PCR}_inv(AIK)) =|>
   State' := 4 /\ SK' := H(exp(KEr',X))
               /\ Nb' := new()
               /\ SND({Nb'.exp(G,X)}_SK')

4. State = 4 /\ RCV({Na.Nb'.KEr'.SML'}_SK) =|>
   State' := 6 /\ secret(SK,sk1,{A,B})
               /\ request(A,B,alice_bob_na,Na)
               /\ request(A,B,alice_bob_nb,Nb')
               /\ request(A,B,sk1,SK)

end role

```

Listing A.1: AVISPA source of the verifier's role

The prover receives the send package in state one and generates an own secret part of his Diffie-Helman key by assigning $Y' := \text{new}()$. He also obtains the current *PCR* values, which is abstracted, since HLPSP does not provide any means of directly modelling TPMs. The generated public key is then signed with the *AIK* together with the nonce and the *PCRs*. The *AIK* is represented by $\text{inv}(\text{AIK})$. After sending the attestation response, the prover also computes the secret key *SK* and witnesses the verifier that he is able to compute a valid signature on the challenge.

```

role bob(A, B: agent,
        AIK, Ks: public_key,
        H: function,
        G: text,
        SND, RCV: channel (dy))
played by B def=
  local State : nat,
        Na, Nb: text,
        Y: text,
        PCR: text,
        SK: message,
        KEi: message,
        SML: text

```

```

init State := 1

transition

1. State = 1 /\ RCV(Na'.KEi') =>
   State' := 3 /\ Y' :=new()
               /\ PCR' :=new()
               /\ SML' :=new()
               /\ SND({B.AIK}_inv(Ks) . {Na'.exp(G, Y') . PCR'}_inv(AIK)
               /\ SK' := H(exp(KEi, Y'))
               /\ witness(B, A, alice_bob_na, Na')

3. State = 3 /\ RCV({Nb'.KEi}_SK) =>
   State' := 5 /\ SND({Na.Nb'.exp(G, Y) . SML'}_SK)
               /\ secret(SML, sml, {A, B})
               /\ secret(SK, sk1, {A, B})
               /\ witness(B, A, sk1, SK)
               /\ witness(B, A, alice_bob_nb, Nb')

end role

```

Listing A.2: AVISPA source of the TPM's role

We use the Dolev-Yao intruder model [50] to model the attacker and the environment. In this intruder model the attacker has full control over all messages that are sent over the network. The attacker can therefore intercept, analyze or modify messages, as well as compose new messages and sent the messages to whoever he pleases. The channels are named *SA*, *RA*, *SB*, *RB*, which stands for send/receive alice and send/receive bob respectively. To abstract away the negotiation of a common generator *g* and a common group *m*, we assume that these are global parameters. These parameters are modelled with the variable *G* and also known to an potential attacker.

```

role session(A, B: agent, Ka, AIK, Ks: public_key, H: function, G:
  text) def=

  local SA, RA, SB, RB: channel (dy)

  composition

    alice(A, B, Ka, Ks, H, G, SA, RA)
    /\ bob (A, B, AIK, Ks, H, G, SB, RB)

  end role

```

```

role environment() def=

    const a, b      : agent,
           aik, ka, ks, ki : public_key,
           h : function,
           g: text,
           na, nb,
           alice_bob_nb,
           bob_alice_na, sec_a_SK, sk1, nb1 : protocol_id

    intruder_knowledge = {a, b, aik, ka, ks, ki, inv(ki), {i.ki}_(inv
                        (ks)), g, h}

    composition

        session(a, b, ka, aik, ks, h, g)
        /\ session(a, i, ka, ki, ks, h, g)
        /\ session(i, b, ki, aik, ks, h, g)

end role

    goal

        secrecy_of sk1
        secrecy_of sml
        authentication_on sk1
        authentication_on alice_bob_na
        authentication_on alice_bob_nb

    end goal

environment()

```

Listing A.3: AVISPA source of environment and security goals

A.2 Protocol 3.7.4

PROTOCOL: Trusted-DHE-TLS: Trusted Transport Layer Security

PURPOSE: Trusted-TLS is intended to provide integrity reporting, privacy and data integrity of communication over the Internet.

The protocol proceeds between a server (role Alice) and a client (role Bob) with respective public keys K_a and K_b . The server wants to securely validate the platform configuration of the client. We assume the existence of one certification authority with public key K_s . This certification authority also created the AIK-Credential. In essence, K_s also acts as privacy CA. The agents possess certificates of the form $\{X, K_x\}_{\text{inv}(K_s)}$. Each session is identified by a unique ID Sid . The protocol also makes use of a pseudo-random number generator PRF which we model as a hash function.

```

role bob(A, B : agent,
        H, PRF, KeyGen: function,
        G:text,
        Kb, Ks, Kaik : public_key,
        SND, RCV: channel (dy))
played_by B def=
  local Na, Nb, Sid, Pb: text,
        State: nat,
        Y:text,
        AIK:text,
        Ka: public_key,
        PCR:text,
        SML:text,
        DHa, DHb, ClientK, ServerK, SK:message

  init State := 0

  transition

  1. State = 0 /\ RCV(start) =|>
      State' := 2
          /\ Nb' := new()
          /\ Pb' := new()
          /\ Sid' := new()
          /\ SND(B.Nb'.Sid'.Pb')

  2. State = 2 /\ RCV(A.Na'.Sid'.Pb'.{A.Ka'}\inv(Ks).{DHa'.Na'.Nb'
    '\inv(Ka')}) =|>
      State' := 4
          /\ Y' := new()
          /\ PCR' := new()
          /\ SML' := new()
          /\ SK' := H(exp(DHa', Y'))
          %% Compute Master Secret
          /\ ClientK' := PRF(SK'.Na'.Nb')
          %% client certificate exchange

```

```

/\ SND ({B.Kb}\_inv(Ks).{AIK.Kaik}\_inv(ks).
%% client key exchange
exp(G,Y').
%% client certificate verify
{H(exp(G,Y').Na'.Nb')}\_inv(Kb).
%% client certificate verify
{H(exp(G,Y').Na'.Nb'.PCR')}\_inv(Kaik)
)
/\ witness(B,A,na\_nbl,Na'.Nb)

3. State = 4 /\ RCV ({DHb'.DHa'.Na'.Nb'.A.B.Na'.Pb'.Sid}\_ClientK)
=>
    %% Send client Finished messages
    State' := 6 /\
        SND ({SML.DHa'.exp(G,Y').Na'.Nb.
        B.A.Na'.Pb'.Sid'}\_ClientK)
        /\ witness(B,A,skl,ServerK')
        /\ secret(ClientK,sec\_clientk,{A,B})
        /\ secret(SML,sml,{A,B})

end role

```

Listing A.4: AVISPA source of the client's role

```

role alice(A, B : agent,
    H, PRF, KeyGen: function,
    G: text,
    Ka, Ks : public_key,
    SND, RCV: channel (dy))
played_by A def=
    local Na, Sid, Pb, PMS: text,
    Nb: text,
    State: nat,
    AIK:text,
    Kb: public_key,
    Kaik: public_key,
    PCR:text,
    X: text,
    SML: text,
    DHa,DHb,ClientK, ServerK,SK:message

    init State := 1

    transition

```

```

1. State = 1 /\ RCV(B.Nb'.Sid'.Pb') =>
   State' := 3
   /\ Na' := new()
   /\ X' := new()
   /\ SND(A.Na'.Sid'.Pb'.{A.Ka}\_inv(Ks).{exp(G,X').Na
     '.Nb'}\_inv(Ka)) %% server hello / certificate
     exchange / server key exchange
   /\ witness(A,B,na\_nb2,Na'.Nb')

   % We simply assume that the server must send back
   % Pa. (Essentially
   % modelling that the client makes only one offer.)

2. State = 3 /\ RCV({B.Kb'}\_inv(Ks)).{AIK.Kaik'}\_inv(ks). %%
   receive client certificate exchange
   %% receive client key exchange
   DHb'.
   %% receive client certificate verify
   {H(DHb'.Na.Nb')}\_inv(Kb').
   %% receive client certificate verify
   {H(DHb'.Na.Nb'.PCR')}\_inv(Kaik')
   ) =>
   State' := 5
   /\ SK' := H(exp(DHb',X'))
   %% Compute Pre\_Master Secret
   /\ ServerK' := PRF(SK'.Na'.Nb')
   %% Compute Master Secret
   /\ SND({DHb'.exp(G,X').Na'.Nb'.
     A.B.Na'.Pb'.Sid'}\_ServerK')
   /\ witness(B,A,sk1,ServerK')

3. State = 5 /\ RCV({SML'.DHa'.DHb'.Na.Nb.B.A.Na.Pb'.Sid'}\_
   _ServerK) => %% Receive client finished message
   State' := 7 /\ request(A,B,na\_nbl,Na.Nb)
   /\ secret(ClientK,sec\_clientk,{A,B})
   /\ secret(ServerK,sec\_serverk,{A,B})

end role

```

Listing A.5: AVISPA source of the Server's role

We again use the Dolev-Yao intruder model [50] to model the attacker and the environment. The channels are again named *SA*, *RA*, *SB*, *RB*, which stands for send/receive alice and send/receive bob respectively.

```

role session(B,A: agent, Ka, Kb, Ks, Kaik: public\_key, H, PRF,
  KeyGen: function, G: text) def=

  local SB, SA, RB, RA: channel (dy)

  composition
    bob(A,B,H,PRF,KeyGen,G,Kb,Ks,Kaik,SB,RB)
    /\ alice(A,B,H,PRF,KeyGen,G,Ka,Ks,SA,RA)

end role

role environment()
def=

  const na\_nb1, na\_nb2, sec\_clientk, sec\_serverk, sml : protocol
    \_id,
    h, prf, keygen : function,
    a, b           : agent,
    ka, kb, ki, ks,kaik : public\_key,
    g              : text

  intruder\_knowledge = { a, b, ka, kb, ks, ki, inv(ki),
    {i.ki}\_-(inv(ks)),g}

  composition
    session(b,a,ka,kb,ks,kaik,h,prf,keygen,g)
    /\ session(b,i,kb,ki,ks,kaik,h,prf,keygen,g)
    /\ session(i,a,ki,ka,ks,kaik,h,prf,keygen,g)

end role

goal

  secrecy\_of sec\_clientk,sec\_serverk

  %Alice authenticates Bob on na\_nb1
  authentication_on na_nb1

  %Bob authenticates Alice on na\_nb2
  authentication_on na_nb2

  %Alice verifies that Bob's TPM was involved in establishing
  the secret communication channel
  authentication_on sk1

  secrecy_of sml

```



```
end goal  
  
environment ()
```

Listing A.6: AVISPA source of the environment and security goals

Scientific Career

03/2009 -

Fraunhofer Institute for Secure Information Technology, Munich
Research Staff Member

03/2008 - 02/2009

Technische Universität Darmstadt, Darmstadt
Research Assistant at IT-Security Group of Prof. Dr. Claudia Eckert

03/2006 - 02/2008

Technische Universität Darmstadt, Darmstadt
Research Assistant at the DFG-Project *TrustCaps*

10/2000 - 2/2006

Technische Universität Darmstadt, Darmstadt
Studies in Computer Science. Graduated as Diplom-Informatiker

Bibliography

- [1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, New York, NY, USA, 2006. ACM.
- [2] Ian Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A Survey on Sensor Networks. *IEEE Comm. Mag.* 40, Issue 8, pages 102–114, 2002.
- [3] Ammar Alkassar and Christian Stübke. Towards Secure IFF: Preventing Mafia Fraud Attacks. In *Proceedings of the 2002 Military Communications Conference (MILCOM 2002)*., 2002.
- [4] A. Alsaïd and C. J. Mitchell. Preventing Phishing Attacks using Trusted Computing Technology. In *INC 2006: Sixth International Network Conference*, pages pp.221–228, Plymouth, UK, July 2006.
- [5] Tiago Alves and Don Felton. TrustZone: Integrated Hardware and Software Security. Technical report, ARM, 2004.
- [6] AMD. AMD Secure Virtual Machine Architecture Reference Manual. Technical report, 2005.
- [7] Steven Hand Andrew Warfield, Keir Fraser and Tim Deegan. Facilitating the Development of Soft Devices. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.
- [8] Applied Data Security Group, University of Bochum. TrustedGRUB. http://www.prosecco.rub.de/trusted_grub_details.html, May 2006.
- [9] C. Richard Attanasio, Peter W. Markstein, and Ray J. Phillips. Penetrating an Operating System: A Study of VM/370 Integrity. *IBM Systems Journal*, 15:102–116, 1976.
- [10] AVISPA. Deliverable 2.3: The High-Level Protocol Specification Language. Technical report, <http://www.avispa-project.org/delivs/2.1/d2-1.pdf>, 2003.

- [11] Shane Balfe, Amit D. Lakhani, and Kenneth G. Paterson. Trusted Computing: Providing Security for Peer-to-Peer Networks. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, 2005.
- [12] Shane Balfe and Kenneth G. Paterson. Augmenting Internet-based Card Not Present Transactions with Trusted Computing: An Analysis. Technical Report RHUL-MA-2006-9, Department of Mathematics, Royal Holloway, University of London, 2005. <http://www.rhul.ac.uk/mathematics/techreports>.
- [13] Shane Balfe and Kenneth G. Paterson. e-EMV: Emulating EMV for Internet payments using Trusted Computing Technology. Technical Report RHUL-MA-2006-10, Department of Mathematics, Royal Holloway, University of London, 2006.
- [14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
- [15] Victor R. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Commun. ACM*, 27(1):42–52, 1984.
- [16] David Basin, Sebastian Mödersheim, Contact Information, and Luca Viganò. OFMC: A Symbolic Model Checker for Security Protocols. *International Journal of Information Security*, 4(3):181–208, June 2004.
- [17] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, 2005.
- [18] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the first ACM Conference on Computer and Communications Security (CCS 1993)*. ACM Press, 1993.
- [19] Muli Ben-Yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, Jun Nakajima, and Elsie Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *Proceedings of the 2006 Ottawa Linux Symposium (OLS 2006)*, 2006.
- [20] Stefan Berger, Ramón Caceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.
- [21] Thomas Beth and Yvo Desmedt. Identification Tokens - or: Solving The Chess Grandmaster Problem. In *Advances in Cryptology-CRYPTO 90*, 1990.
- [22] Colin Boyd and Anish Mathuria. *Protocols for Authentication and Key Establishment*. Springer-Verlag, 2003.

- [23] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct Anonymous Attestation. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 132–145, New York, NY, USA, 2004. ACM Press.
- [24] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating Systems Principles*, pages 143–156, New York, NY, USA, 1997. ACM Press.
- [25] Bundesnetzagentur. Statement for Electronic Signatures No. 58, 2006.
- [26] Bundestags-Drucksache 14/4662, 21, 29, 30. Official Statement to SigVO, 27, 2001.
- [27] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.
- [28] Jan Camenisch and Anna Lysyanskaya. A Signature Scheme with Efficient Protocols. In *International Conference on Security in Communication Networks – SCN*, volume 2576 of *Lecture Notes in Computer Science*, pages 268–289. Springer Verlag, 2002.
- [29] Patrick Carroll. Study of Buffer Overflows and Keyloggers in the Linux Operating System. Technical report, Computer Science Department University of Maryland, Baltimore County, 2007.
- [30] Katharine Chang and Kang G. Shin. Distributed Authentication of Program Integrity Verification in Wireless Sensor Networks. *ACM Transactions on Information and System Security*, 11(3):1–35, 2008.
- [31] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, Christian Stueble, and Horst Görtz. A Protocol for Property-based Attestation. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, New York, NY, USA, 2006. ACM Press.
- [32] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.
- [33] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drieslma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Proceedings of Workshop on Specification and Automated Processing of Security Requirements (SAPS 2004)*, 2004.
- [34] SuGil Choi, JinHee Han, and SungIk Jun. Improvement on TCG Attestation and Its Implication for DRM. In *Proceedings of Computational Science and Its Applications - ICCSA 2007*, Lecture Notes in Computer Science, pages 912–925. Springer Verlag, August 2007.

- [35] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM Press.
- [36] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with brutus. volume 9, pages 443–487, New York, NY, USA, 2000. ACM.
- [37] Intel Corporation. Intel virtualization technology for directed i/o architecture specification. Technical report, Intel, 2006.
- [38] Intel Corporation. Intel(r) trusted execution technology preliminary architecture specification. Technical report, November 2006.
- [39] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide. Technical report, Intel, February 2008.
- [40] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A Safety-Oriented Platform for Web Applications. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 350–364, Washington, DC, USA, 2006. IEEE Computer Society.
- [41] Crossbow Technology. Mica2 Datasheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf.
- [42] Crossbow Technology. Stargate Datasheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Stargate_Datasheet.pdf.
- [43] DGE Inc. Gateway 3 (can/lin/j1850 gateway). Technical report, DGE Inc., 2006.
- [44] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590, New York, NY, USA, 2006. ACM.
- [45] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. Technical report, IETF Network Working Group, 2006.
- [46] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
- [47] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society, 1992.
- [48] Thomas Dohmke. Bussysteme im Automobil CAN, FlexRay und MOST. Technical report, TU Berlin, March 2002.

- [49] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable cryptography. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 542–552, New York, NY, USA, 1991. ACM.
- [50] Danny Dolev and Andrew C. Yao. On the Security of Public Key Protocols. In *Proceedings of the IEEE 22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, 1981.
- [51] Xin Dong. Attestation over Multiple Hops in Hybrid Wireless Sensor Networks. Master's thesis, TU Darmstadt, 2008.
- [52] Yaozu Dong, Shaofan Li, Asit Mallick, Jun Nakajima, Kun Tian, Xuefei Xu, Fred Yang, and Wilfred Yu. Extending Xen with Intel Virtualization Technology. *Intel Technology Journal*, August 2006. <http://www.intel.com/technology/itj/2006/v10i3/>.
- [53] Loïc Dufлот and Laurent Absil. Programmed I/O Accesses: A Threat to Virtual Machine Monitors? In *In Proceedings of the fifth annual PacSec Applied Security Conference*, 2007.
- [54] Claudia Eckert. *IT Sicherheit - Konzepte, Verfahren, Protokolle, 5. Auflage*. R. Oldenbourg Verlag, 4 edition, October 2008.
- [55] EMSCB. Towards Trustworthy Systems with Open Standards and Trusted Computing. <http://www.emscb.de/>, 2006.
- [56] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, 2003.
- [57] Paul England and Jork Loeser. Para-Virtualized TPM Sharing. In *In Proceedings of the First International Conference on Trusted Computing and Trust in Information Technologies (Trust 2008)*, number 4968 in Lecture Notes in Computer Science, pages 119–132. Springer-Verlag, 2008.
- [58] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating Systems Principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.
- [59] EuPD Research. eCommerce 2007. Technical report, 2007.
- [60] Peter Ferrie. Attacks on Virtual Machine Emulators. Technical report, Symantec Advanced Threat Research, 2007.
- [61] Renato J. O. Figueiredo, Peter A. Dinda, and José A. B. Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(5):28–31, 2005.

- [62] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004.
- [63] Andreas Fuchs. Improving scalability of remote attestation. Master's thesis, Technische Universität Darmstadt, 2008.
- [64] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally Composable Security Analysis of TLS. In *Proceedings of the 2nd International Conference on Provable Security (ProvSec 2008)*, volume Lecture Notes in Computer Science. Springer, 2008.
- [65] Sebastian Gajek, Ahmad-Reza Sadeghi, Christian Stübke, and Marcel Winandy. Compartmented Security for Browsers - Or How to Thwart a Phisher with Trusted Computing. In *In Proceedings of the Second International Conference on Availability, Reliability and Security (ARES 2007)*, 2007.
- [66] Saurabh Ganeriwal, Srivaths Ravi, and Anand Raghunathan. Trusted Platform based Key Establishment and Management for Sensor Networks. 2007. Under review.
- [67] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating Systems Principles*, pages 193–206, New York, NY, USA, 2003. ACM Press.
- [68] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, and N. Asokan. Beyond Secure Channels. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 30–40, New York, NY, USA, 2007. ACM.
- [69] Robert P. Goldberg. Survey of Virtual Machine Research. *Computer*, June 1974.
- [70] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking Remote Attestation to Secure Tunnel Endpoints. In *First ACM Workshop on Scalable Trusted Computing*, Fairfax, Virginia, November 2006.
- [71] Li Gong, Roger Needham, and Raphael (1990) Reasoning About Belief in Cryptographic Protocols. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1990.
- [72] Greg Goth. Phishing Attacks Rising, But Dollar Losses Down. volume 3, page 8, Piscataway, NJ, USA, 2005. IEEE Educational Activities Department.
- [73] David Grawrock. *The Intel Safer Computing Initiative - Building Blocks for Trusted Computing*. Intel Press, 2006.

- [74] Sigrid Gürgens, Peter Ochsenschläger, and Carsten Rudolph. Role based specification and security analysis of cryptographic protocols using asynchronous product automata. In *DEXA 2002 International Workshop on Trust and Privacy in Digital Business*, 2002.
- [75] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing. In *USENIX Virtual Machine Research and Technology Symposium*, 2004 2004.
- [76] Steven Hand, Andrew Wafield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *ACM OASIS Workshop*, 2005.
- [77] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, January 2006.
- [78] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System. In *EDCC '06: Proceedings of the Sixth European Dependable Computing Conference*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
- [79] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, 2006.
- [80] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. volume 40, pages 80–89, New York, NY, USA, 2006. ACM.
- [81] Martin Hermanowski. Eine auf Virtualisierung beruhende vertrauenswürdige Architektur für Geschäftsprozesse. Master’s thesis, TU Darmstadt, 2007.
- [82] Intel Corporation. 82527 Serial Communications Controller. Technical report, Intel Automotive, 2004.
- [83] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28, New York, NY, USA, 2006. ACM Press.
- [84] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, August 2003.
- [85] Markus Jakobsson and Moti Yung. Distributed ”Magic Ink” Signatures. In *Advances in Cryptology - Proceedings of Eurocrypt '97*, volume 1233, pages 450–??, 1997.

- [86] Bernhard Jansen, Hari-Govind V. Ramasamy, and Matthias Schunter. Policy Enforcement and Compliance Proofs for Xen Virtual Machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110, New York, NY, USA, 2008. ACM.
- [87] Bernhard Jansen, HariGovind Ramasamy, and Matthias Schunter. Flexible Integrity Protection and Verification Architecture for Virtual Machine Monitors. In *Second Workshop on Advances in Trusted Computing*, 2006.
- [88] Audun. Jøsang, M. Patton, and A. Ho. Authentication for Humans. In *Proceedings of the 9th International Conference on Telecommunication Systems (ICTS2001)*, March 2001.
- [89] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A Retrospective on the VAX VMM Security Kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.
- [90] Chris Karlof and David Wagner. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. In *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*, pages 113–127, 2003.
- [91] Stefan Katzenbeisser, Klaus Kursawe, and Frederic Stumpf. Revocation of TPM Keys. In *Proceedings of the Second International Conference on Trusted Computing (TRUST 2009)*, Lecture Notes in Computer Science. Springer-Verlag, 2009.
- [92] Bernhard Kauer. OSLO: Improving the Security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*, August 2007.
- [93] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.
- [94] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 Annual USENIX Technical Conference*, pages 71–84, June 2003.
- [95] Rafal Kolanski and Gerwin Klein. Formalising the L4 microkernel API. In *CATS '06: Proceedings of the 12th Computing: The Australasian Theroy Symposium*, pages 53–68, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
- [96] Christoph Krauß, Frederic Stumpf, and Claudia Eckert. Detecting Node Compromise in Hybrid Wireless Sensor Networks Using Attestation Techniques. In *Proceedings of the Fourth European Workshop on Security and Privacy in Ad hoc and Sensor Networks*, volume 4572 of *Lecture Notes in Computer Science*, pages 203–217, Cambridge, UK, July 2007. ESAS2007, Springer-Verlag.

- [97] Nicolai Kuntze and Andreas U. Schmidt. Protection of DVB Systems by Trusted Computing. In *IEEE International Symposium on Broadband Multimedia Systems and Broadcasting 2007, 28-29 March 2007 at the Orange County Convention Center, Orlando, FL, USA*, 2007.
- [98] Leslie Lamport. Password Authentication with Insecure Communication. In *Commun. ACM*, volume 24, pages 770–772, New York, NY, USA, 1981. ACM Press.
- [99] Hanno Langweg. Malware Attacks on Electronic Signatures Revisited. In *Tagungsband Sicherheit 2006*, pages 244–255, 2006.
- [100] Susan C. Lee and Lauren B. Davis. Learning from Experience: Operating System Vulnerability Trends. volume 5, pages 17–24, Piscataway, NJ, USA, 2003. IEEE Computer Society.
- [101] J. Liedtke. On Micro-Kernel Construction. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 237–250, New York, NY, USA, 1995. ACM Press.
- [102] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996.
- [103] Hans Löhr, HariGovind V. Ramasamy, Ahmad-Reza Sadeghi, Stefan Schulz, Matthias Schunter, and Christian Stübke. Enhancing Grid Security Using Trusted Virtualization. In *Second Workshop on Advances in Trusted Computing (WATC'06)*, Tokyo, Japan, December 2006.
- [104] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using *fd*. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.
- [105] Stuart E. Madnick and John J. Donovan. Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 210–224, New York, NY, USA, 1973. ACM Press.
- [106] John Marchesini, Sean W. Smith, Omen Wild, and Rich MacDonald. Experimenting with TCGA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report Technical Report TR2003-476, Department of Computer Science, Dartmouth College, December 2003.
- [107] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Minimal TCB Code Execution. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 267–272, Washington, DC, USA, 2007. IEEE Computer Society.
- [108] Catherine Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, Volume 9, Numbers 1-2/2001, 2001.

- [109] Catherine A. Meadows. Formal verification of cryptographic protocols: A survey. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag, 1994.
- [110] OpenTC. Open Trusted Computing (OpenTC). <http://www.opentc.net/>, 2007.
- [111] Rolf Oppliger and Sebastian Gajek. Effective Protection Against Phishing and Web Spoofing. In *Proceedings of the 9th IFIP TC6 and TC11 Conference on Communications and Multimedia Security (CMS 2005)*, LNCS 3677, pages pp. 32–41. Springer-Verlag, September 2005.
- [112] Rolf Oppliger, Ralf Hauser, and David Basin. SSL/TLS Session-Aware User Authentication — Or How to Effectively Thwart the Man-in-the-Middle. *Computer Communications*, 29:2238–2246, 2006.
- [113] Christoph Paar, Andre Weimerskirch, and Marko Wolf. Security in Automotive Bus Systems. In *Embedded Security in Cars Workshop (escar 2004)*, 2004.
- [114] Taejoon Park and Kang G. Shin. Soft Tamper-Proofing via Program Integrity Verification in Wireless Sensor Networks. *IEEE Transactions on Mobile Computing*, 4(3):297–309, 2005.
- [115] Lawrence C. Paulson. Mechanized proofs of security protocols: Needham-schroeder with public keys. Technical report, University of Cambridge, 1997.
- [116] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6(3):85–128, 1998.
- [117] Lawrence C. Paulson. Inductive Analysis of the Internet Protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.
- [118] Marcus Peinado, Yuqun Chen, Paul England, and John Manferdell. NGSCB: A Trusted Open System. In *Proceedings of the 9th Australasian Conference (ACISP 2004)*, Lecture Notes in Computer Science, pages 86–97. Springer Berlin, 2004.
- [119] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. SPINS: Security Protocols for Sensor Networks. *Wirel. Netw.*, 8(5):521–534, 2002.
- [120] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. SPINS: security protocols for sensor networks. *Wirel. Netw.*, 8(5):521–534, 2002.
- [121] Birgit Pfitzmann, James Riordan, Christian Stübke, Michael Waidner, and Arnd Weber. The PERSEUS System Architecture. In *VIS 2001, Sicherheit in komplexen IT-Infrastrukturen*. Vieweg Verlag, 2001.
- [122] PHILIPS. 32-bit ARM7 Microcontroller with IVN Gateway Functionality. Technical report, PHILIPS, 2003.

- [123] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
- [124] Gerald J. Popek and Charles S. Kline. A Verifiable Protection System. In *Proceedings of the international conference on Reliable software*, pages 294–304, New York, NY, USA, 1975. ACM.
- [125] Jason F. Reid and William J. Caelli. DRM, Trusted Computing and Operating System Architecture. In *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid Computing and E-Research*, pages 127–136, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
- [126] European Commision Press Release. Electronic Signatures: Legally Recognised but Cross-Border Take-Up to Slow. IP/06/325.17/03/2006, 2006.
- [127] Ronald L. Rivest. The RC5 Encryption Algorithm. In *Proceedings of the 1994 Leuven Workshop on Fast Software Encryption*, pages 86–96. Springer, 1995.
- [128] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 26(1):96–99, 1983.
- [129] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium, Denver, CO*, August 2000.
- [130] Patrick Röder, Frederic Stumpf, Ralf Grewe, and Claudia Eckert. Hades - Hardware Assisted Document Security. In *Proceedings of the Second Workshop on Advances in Trusted Computing (WATC'06)*, Tokyo, Japan, November 2006.
- [131] Phillip Rogaway and Thomas Shrimpton. *Fast Software Encryption*, chapter Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. LNCS. Springer, 2004.
- [132] Mendel Rosenblum and Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *Computer*, July 2005.
- [133] W. M. Row, Donald. J. Morton, B. L. Adams, and A. H. Wright. Security Issues in Small Linux Networks. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 506–510, New York, NY, USA, 1999. ACM Press.
- [134] Alexander Roßnagel and Stefanie Fischer-Dieskau. Elektronische Dokumente als Beweismittel - Neufassung der Beweisregelungen durch das Justizkommunikationsgesetz. *Neue Juristische Wochenschrift (NJW)*, 59:806–809, 2006.
- [135] Joanna Rutkowska. Subverting Vista Kernel For Fun And Profit. Black Hat Briefings 2006, August 2006.

- [136] Ahmad-Reza Sadeghi. Trusted Computing - Special Aspects and Challenges. *SOFSEM 2008: Theory and Practice of Computer Science*, pages 98–117, 2008.
- [137] Ahmad-Reza Sadeghi, Michael Scheibel, Christian Stübke, and Marco Wolf. Play it once again, Sam - Enforcing Stateful Licenses on Open Platforms. In *2nd Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, November 2006.
- [138] Ahmad-Reza Sadeghi, Marcel Selhorst, Christian Stübke, and Marcel Winandy. TCG Inside? A Note on TPM Specification Compliance. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing (STC'06)*., 2006.
- [139] Ahmad-Reza Sadeghi and Christian Stueble. Property-based Attestation for Computing Platforms: Caring about Properties, not Mechanisms. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, New York, NY, USA, 2004. ACM Press.
- [140] Reiner Sailer, Trent Jaeger, , Enriquillo Valdez, Ramon Caceres, and Ronald Perez. Building a mac-based security architecture for the xen opensource hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.
- [141] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based Policy Enforcement for Remote Access. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317, New York, NY, USA, 2004. ACM Press.
- [142] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th USENIX Security Symposium*. IBM T. J. Watson Research Center, August 2004.
- [143] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, pages 1278–1308. IEEE Computer Society, 1975.
- [144] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 17th USENIX conference on System administration (LISA '03)*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.
- [145] Evan Sparks Sergey Bratus, Nihal D'Cunha and Sean W. Smith. TOCTOU, Traps, and Trusted Computing. In *TRUST2008*, 2008.
- [146] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Externally verifiable code execution. *Commun. ACM*, 49(9):45–49, 2006.

- [147] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure Code Update By Attestation in sensor networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, New York, NY, USA, 2006. ACM Press.
- [148] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, 2007.
- [149] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2005. ACM Press.
- [150] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.
- [151] Vincent Y. Shen, Tze-Jie Yu, Stephen M. Thebaut, and Lorri R. Paulsen. Identifying error-prone software an empirical study. In *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, volume 11, pages 317–324, Piscataway, NJ, USA, 1985. IEEE Press.
- [152] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168, Washington, DC, USA, 2005. IEEE Computer Society.
- [153] R. Shirey. Internet security glossary. Technical report, IETF Network Working Group, 2000.
- [154] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 161–174, New York, NY, USA, 2006. ACM Press.
- [155] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, 1993.
- [156] Frederic Stumpf, Michael Benz, Martin Hermanowski, and Claudia Eckert. An Approach to a Trustworthy System Architecture using Virtualization. In *Proceedings of the 4th International Conference on Autonomic and Trusted Computing (ATC-2007)*, volume 4158 of *Lecture Notes in Computer Science*, pages 191–202, Hong Kong, China, 2007. Springer-Verlag.

- [157] Frederic Stumpf and Claudia Eckert. Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques. In *Proceedings of the Second International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2008)*, pages 1–9, Cap Esterel, France, August 25-31 2008. IEEE Computer Society.
- [158] Frederic Stumpf, Claudia Eckert, and Shane Balfe. Towards Secure E-Commerce Based on Virtualization and Attestation Techniques. In *Proceedings of the Third International Conference on Availability, Reliability and Security (ARES 2008)*, pages 376–382, Barcelona, Spain, March 4 - 7 2008. IEEE Computer Society.
- [159] Frederic Stumpf, Lars Fischer, and Claudia Eckert. Trust, Security and Privacy in VANETs – A Multilayered Security Architecture for C2C-Communication. In *VDI/VW-Gemeinschaftstagung: Automotive Security*, Wolfsburg, Germany, November 2007.
- [160] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the Scalability of Platform Attestation. In *Proceedings of the Third ACM Workshop on Scalable Trusted Computing (ACM STC'08)*, pages 1–10, Fairfax, USA, October 31 2008. ACM Press.
- [161] Frederic Stumpf, Patrick Röder, and Claudia Eckert. An Architecture Providing Virtualization-Based Protection Mechanisms against Insider Attacks. In *Proceedings of the 8th International Workshop on Information Security Applications (WISA 2007)*, volume 4867 of *Lecture Notes in Computer Science*, pages 142–156, Jeju Island, Korea, 2007. Springer-Verlag.
- [162] Frederic Stumpf, Markus Sacher, Alexander Roßnagel, and Claudia Eckert. Erzeugung elektronischer Signaturen mittels Trusted Platform Module. *Datenschutz und Datensicherheit*, 4:357–361, April 2007.
- [163] Frederic Stumpf, Markus Sacher, Alexander Roßnagel, and Claudia Eckert. The Creation of Qualified Signatures with Trusted Platform Modules. *Digital Evidence Journal*, 4(No. 2):81–86, 2007.
- [164] Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. A Robust Integrity Reporting Protocol for Remote Attestation. In *Second Workshop on Advances in Trusted Computing (WATC'06 Fall)*, Tokyo, Japan, November 2006.
- [165] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.
- [166] Andrew S. Tanenbaum. *Modern Operating Systems*, volume 3. Pearson Prentice Hall, 2008.
- [167] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can We Make Operating Systems Reliable and Secure? *Computer*, 39(Issue 5):44–51, May 2006.

- [168] Trusted Computing Group . TCG TPM Specification Version 1.2 Revision 103, TPM Main Part 2, TPM Structures. Technical report, TCG, July 2007.
- [169] Trusted Computing Group. Infrastructure Subject Key Attestation Evidence Extension Version 1.0, Revision 5. Technical report, TCG, 2005.
- [170] Trusted Computing Group. TCG PC Client Specific Implementation Specification for Conventional BIOS. Technical report, July 2005.
- [171] Trusted Computing Group. TCG TSS Specification Version 1.2. Technical report, TCG, 2006.
- [172] Trusted Computing Group. TCG Mobile Trusted Module Specification, Revision 1.0. Technical report, TCG, July 2007.
- [173] Trusted Computing Group. TCG TPM Specification, Architecture Overview. Technical report, TCG, 2007.
- [174] Trusted Computing Group. TCG TPM Specification Version 1.2 Revision 103, Design Principles. Technical report, TCG, July 2007.
- [175] Trusted Computing Group. TCG TPM Specification Version 1.2 Revision 103, TPM Main Part 3 Commands. Technical report, TCG, July 2007.
- [176] Trusted Computing Group. TCG Trusted Network Connect TNC Architecture for Interoperability. Technical report, TCG, 2007.
- [177] Trusted Computing Group. TCG Trusted Network Connect TNC IF-IMV. Technical report, TCG, 2007.
- [178] Trusted Computing Group. Replacing Vulnerable Software with Secure Hardware. Technical report, TCG, 2008.
- [179] Trusted Computing Group. Trusted Platform Module (TPM) Specifications. Technical report, <https://www.trustedcomputinggroup.org/specs/TPM>, 2008.
- [180] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando Martins, Andrew Anderson, Steven Bennett, Alain Kaegi, Felix Leung, and Larry Smith. Intel Virtualization Technology. *IEEE Computer Society*, 5:48–56, July 2005.
- [181] Martin Unger, Ahmad-Reza Sadeghi, Gianluca Ramunno, Davide Vernizzi, Yacine Gasmi, Patrick Stewin, and Frederik Armknecht. An Efficient Implementation of Trusted Channels based on OpenSSL. In *Proceedings of the Third ACM Workshop on Scalable Trusted Computing (ACM STC'08)*. ACM Press, October 2008.
- [182] Luca Viganò. Automated Security Protocol Analysis with the AVISPA Tool. In *Proceedings of the XXI Mathematical Foundations of Programming Semantics (MFPS'05)*, volume 155 of *ENTCS*, pages 61–86. Elsevier, 2005.

- [183] VMWare. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. Technical report, VMware, 2007.
- [184] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 Protocol. In *Second USENIX Workshop on Electronic Commerce*, 1996.
- [185] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the 5th USENIX Symposium on Operating Systems*, 2002.
- [186] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection Strategies for Direct Access to Virtualized I/O Devices. In *In Proceedings of the 2008 USENIX Annual Technical Conference*, 2008.
- [187] Rafal Wojtczuk. Subverting the Xen Hypervisor. Black Hat USA 2008, August 2008.
- [188] Min Wu, Robert C. Miller, and Greg Little. Web Wallet: Preventing Phishing Attacks by Revealing User Intentions. In *SOUPS '06: Proceedings of the second symposium on Usable privacy and security*, pages 102–113, New York, NY, USA, 2006. ACM.
- [189] Xen. Xen 3.0 User Manual. Technical report, Xen Community, 2007.
- [190] Fan Ye, Haiyun Luo, Songwu Lu, and Lixia Zhang. Statistical En-Route Filtering of Injected False Data in Sensor Networks. In *Proceedings IEEE INFOCOM*, 2004.
- [191] Sencun Zhu, Sanjeev Setia, and Sushil Jajodia. LEAP: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 2003.
- [192] Dino A. Dai Zovi. Hardware Virtualization Rootkits. BlackHat Briefings USA, August 2006, Las Vegas, NV., 2006.